JULIO CESAR GOLDNER VENDRAMINI

REDE OMEGA VIRTUAL EM FPGA COM RECONFIGURAÇÃO EM TEMPO DE EXECUÇÃO. ESTUDO DE CASO: CÁLCULO DE ATRATORES EM REDES REGULADORAS DE GENES.

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de Magister Scientiae.

VIÇOSA
MINAS GERAIS – BRASIL
2012

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
V453r
2012

Vendramini, Julio Cesar Goldner, 1987-
Rede Omega virtual em FPGA com reconfiguração em tempo de execução: estudo de caso: cálculo de atratores em redes reguladoras de genes / Julio Cesar Goldner Vendramini. – Viçosa, MG, 2012.
101f. : il. ; 29cm.

Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 96-101

1. Arquitetura de computador. 2. Dispositivo de lógica programável. 3. Arranjo de lógica programável em campo. 4. Redes reguladoras de genes. 5. Atratores (Matemática). I. Universidade Federal de Viçosa. II. Título.

CDD 22. ed. 004.22

JULIO CESAR GOLDNER VENDRAMINI

REDE OMEGA VIRTUAL EM FPGA COM RECONFIGURAÇÃO EM TEMPO DE EXECUÇÃO. ESTUDO DE CASO: CÁLCULO DE ATRATORES EM REDES REGULADORAS DE GENES

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 16 de março de 2012.

Henrique Cota de Freitas

Carlos Augusto Paiva da Silva Martins

José Augusto Miranda Nacif

Ricardo dos Santos Ferreira
(Orientador)

AGRADECIMENTOS

Agradeço especialmente à Deus por mais essa vitória. Agradeço aos meus pais pelo carinho e apoio nas horas que necessitei. As minhas irmãs pelas palavras de incentivo. Agradeço ao meu Orientador, Ricardo dos Santos Ferreira, por todo apoio e orientação para conclusão deste trabalho. Agradeço também a CAPES pelo financiamento da bolsa e pela FAPEMIG pelo financiamento do projeto, permitindo assim a aquisição das ferramentas necessárias para desenvolvimento do trabalho aqui realizado.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

ABSTRACT

1. Introdução 1

1.1 Motivação 2

1.2 Proposta 3

1.3 Estrutura do Trabalho 3

2. Revisão Bibliográfica 5

2.1 Redes de Interconexão Multiestágio 5

2.1.1 Classificação das Redes Multiestágios 5

2.1.2 Trabalhos Correlatos 10

2.2 FPGA 13

3. Rede Multiestágio Omega 16

3.1 Metodologia para os testes 16

3.2 Resultados Experimentais 17

3.2.1 Área 18

3.2.2 Latência 19

3.2.3 Bits de Configuração 20

3.2.4 Rede Multiestágio e Unidades Funcionais 23

3.2.5 Registros entre Estágios 24

3.2.6 Radix do Comutador 25

3.2.7 Rede Benes 26

3.2.8 FPGA 27

3.3 Considerações finais 28

4. Roteamento da Rede Omega 29

4.1 Trabalhos Correlatos 29
4.2 O Roteamento 31
4.3 O algoritmo usando memórias embarcadas 34
4.4 O algoritmo usando codificador de prioridade 39
4.5 Resultados experimentais 43
4.5.1 Capacidade de Roteamento Radix2 43
4.5.2 Tentativas nos Estágios Extras Radix2 47
4.5.3 Capacidade de Roteamento Radix4 48
4.5.4 Tentativas nos Estágios Extras Radix4 50
4.5.5 Implementação do Algoritmo de Roteamento em FPGA 51
4.5.6 Capacidade de Roteamento na Presença de Multicast 54
4.6 Considerações Finais 56
5. Algoritmo Escalonamento, Posicionamento e Roteamento em uma Arquitetura CGRA com Multiestágio 57
5.1 Arquitetura reconfigurável proposta 58
5.2 Exemplo de Mapeamento 58
5.3 Escalonamento 59
5.4 Algoritmo Escalonamento, Posicionamento e Roteamento (SPR) 61
5.5 Posicionamento 63
5.6 Roteamento 65
5.7 Resultados Experimentais 66
5.8 Considerações Finais 70
6. Redes reguladoras de genes em FPGA 71
6.1 Modelo Grafo Booleano 72
6.1.2 Grafo Livre de Escala 74
6.1.3 Algoritmo para cálculo do período dos atratores 74
6.1.4 Trabalhos Correlatos 76
6.2 Arquiteturas utilizando FPGA 77

6.3 A arquitetura proposta e Algoritmo Paralelo em Hardware 78

6.4 A síntese em FPGA 81

6.5 Arquitetura e interface com a Placa Virtex6 83

6.6 Resultados Experimentais 86

6.6 Considerações Finais 91

7. Conclusões 92

8. Bibliografia 96

LISTA DE FIGURAS

Figura 2.1: Classificação das Redes de interconexão multiestágio. [Grammatikakis, 2000] 6
Figura 2.2: Permutação de entradas. 7
Figura 2.3: Rede Omega. 8
Figura 2.4: Rede rearranjável Benes 8
Figura 2.5: Rede não bloqueante Clos. 9
Figura 2.6: Comutadores: (a) 2x2; (b) 4x4 (c) Radix4. 9
Figura 2.7: Arquitetura interna do FPGA 14
Figura 2.8: (a) FPGA Virtex-6; (b) placa de protótipo FPGA, Xilinx ML605, utilizada no projeto. 15
Figura 3.1: (a) comutador em rede radix2; (b) comutador em rede radix4. 17
Figura 3.2: Rede Omega 8x8 com largura de 16 bits. 18
Figura 3.3: Rede Omega 16x16, comparativo com rede Omega 8x8. 20
Figura 3.4: Rede Omega com as entradas dos bits de configuração. 21
Figura 3.5: Rede Omega utilizando Memórias para armazenar suas configurações de roteamento. 22
Figura 3.6: Rede Omega de tamanho 64x64 utilizando 6 módulos de memória(BRAM) 22
Figura 3.7: Rede multiestágio interconectando ALUs; (a) síntese em separado; (b) síntese de todo o conjunto. 24
Figura 3.8: Rede multiestágio com registros nas colunas 24
Figura 3.9: Rede Omega 16x16 utilizando radix2 e radix4 25
Figura 4.1: Redes: (a) Omega; (b) Omega com 1 estágio extra. 31
Figura 4.2: Algoritmo de roteamento da rede Omega radix2 com estágios extra. 33
Figura 4.3: Arquitetura do algoritmo utilizando janela em memórias 34
Figura 4.4: (a) Memória de configuração da arquitetura de roteamento – radix2; (b) Funcionamento da arquitetura por janelas usando memórias fazendo o roteamento de uma conexão – radix2. 35
Figura 4.5: (a) Memória de configuração da arquitetura de roteamento – radix4; (b) Funcionamento da arquitetura por janelas usando memórias gerando conflito no rotemento – radix4. 37
Figura 4.6: (a) Arquitetura roteando a ligação 10 -> 3 com extra = 1; (b) Memória com as ligações 5 -> 2 e 10 -> 3. 37
Figura 4.7: Duas Redes Omegas em Paralelo VS. Rede Rearranjável. 38
Figura 4.8: Pseudocódigo algoritmo de roteamento das redes em paralelo. 38

Figura 4.9: Pseudocódigo algoritmo de roteamento das redes sobrecarregando a rede 0. 39
Figura 4.10: *Broadcast* 41
Figura 4.11: Unidades de Controle: (a) Lg n estágios; (b) extras. 41
Figura 4.12: Codificador Prioridade: (a) Parcial (b) 2 Níveis. 42
Figura 4.13: (a) Rede Radix2 com 2 extras, com 50% de carga de trabalho; Duas redes de tamanho 64 radix4 em paralelo com um estágio extra. 44
Figura 4.14: Roteamento rede de tamanho 64, com uma e duas redes, com e sem estágios extras 45
Figura 4.15: Roteamento rede de tamanho 256, com uma e duas redes, com e sem estágios extras 46
Figura 4.16: Roteamento rede de tamanho 512, com uma e duas redes, com e sem estágios extras 46
Figura 4.17: Roteamento rede de tamanho 1024, com uma e duas redes, com e sem estágios extras 47
Figura 4.18: Médias de tentativas nas redes com estágios extras. 47
Figura 4.19: Roteamento rede de tamanho 64, com uma e duas redes, com e sem estágios extras 49
Figura 4.20: Roteamento rede de tamanho 256, com uma e duas redes, com e sem estágios extras 49
Figura 4.21: Roteamento rede de tamanho 1024, com uma e duas redes, com e sem estágios extras 50
Figura 4.22: Médias de tentativas nas redes radix4 com estágios extras. 50
Figura 4.23: Análise roteamento conexões numa radix4 com e sem *multicast* para rede de tamanho 64. 54
Figura 4.24: Médias de tentativas de roteamento por conexão, rede radix4 tamanho 64 55
Figura 5.1: Arquitetura reconfigurável proposta 58
Figura 5.2: Um exemplo simples de mapeamento. 59
Figura 5.3: Um exemplo utilizando *pipeline*. 60
Figura 5.4: Exemplo de utilização de registro para passagem de valor do nó 61
Figura 5.5: Pseudo código do algoritmo de escalonamento, posicionamento e roteamento(SPR) 62
Figura 5.6: Dois escalonamentos do grafo *Motion Vector*. (a) com II igual a 2 (b) com II igual a 3 63
Figura 5.7: Grafo FIR; (a) Ordem do ALAP; (b) SPR 64

Figura 6.1: Rede Booleana de 3 vértices e seu diagrama de estados: (a) Evolução da rede Booleana no tempo (b) Diagrama parcial de transição de estados (c) Diagrama completo de transição com 3 atratores e suas bacias. 73
Figura 6.2: (a) Rede Aleatória com K=2 em todos os vértices; (b) Rede Livre de Escala 74
Figura 6.3: Sequência de passos para cálculo do período com duas simulações S0 e S1 para um diagrama de estados parcial. 75
Figura 6.4: Pseudocódigo para cálculo do Período – Versão Sequencial 76
Figura 6.5: Pseudo-Código para cálculo do Período – Versão Paralela para FPGA 80
Figura 6.6: Modelo do Vértice em hardware 80
Figura 6.7: Diagrama para implementação dos vértices e conexões no FPGA 81
Figura 6.8: Arquitetura Genérica para um grafo com N vértices e um conjunto de arestas reconfigurável 82
Figura 6.9: Geração de vários grafos sem a necessidade de re-sintetizar 82
Figura 6.10: Arquitetura implementada utilizando o Microblaze para comunicação com o computador. 83
Figura 6.11: Arquitetura Completa com mais detalhes 84
Figura 6.12: Arquitetura de cálculo do período e transiente 85
Figura 6.13: Arquitetura responsável por criar as configurações das memórias para o cálculo do período. 86
Figura 6.14: Ganho do algoritmo em hardware contra do em software 90

LISTA DE TABELAS

Tabela 3.1: Tamanho da Rede Multiestágio Omega. Sintetizada em uma Virtex6 XC6VLX240T com 150730 Luts. 18
Tabela 3.2: Latência (ns) Rede Multiestágio Omega. Sintetizada em uma Virtex6 XC6VLX240T. 20
Tabela 3.3: Módulos de Memória sintetizado em uma Virtex6 com 416 módulos..... 23
Tabela 3.4: Área das ALU sintetizadas em separado e em conjunto com uma rede Omega de 32 entradas. 23
Tabela 3.5: Rede com *Pipeline* (Registros entre os estágios). Largura de 8 bits..... 25
Tabela 3.6: Tamanho Rede Radix 4 e Radix 2. Área em LUTs e atraso em ns. 26
Tabela 3.7: Rede Benes. Largura de 8 bits. Área em LUTs e atraso em ns. 27
Tabela 3.8: Custo em área em LUTs para uma Spartan3 e uma Virtex6 com 16 bits de largura.27
Tabela 4.1: Área em LUTs/Memórias e Ciclo de relógio em ns para o Algoritmo de Janelas em Memória..... 51
Tabela 4.2: Área em LUTs e Ciclo de relógio em ns para o Algoritmo com codificador de Prioridade..... 52
Tabela 4.3: Área em LUTs para Codificadores de Prioridade..... 53
Tabela 4.4: Atraso em ns para Codificadores de Prioridade..... 53
Tabela 4.5: Recursos utilizados no FPGA para roteamento por hardware, rede de tamanho 64 com um estágio extra e *multicast*. 55
Tabela 5.1: Características do grafos utilizados..... 66
Tabela 5.2: Características das arquiteturas escolhidas 67
Tabela 5.3: Resultados do SPR para a arquitetura A_1 68
Tabela 5.4: Resultados do SPR para a arquitetura A_6 68
Tabela 5.5: Arquiteturas mapeadas em FPGA, Xilinx Virtex6. 69
Tabela 5.6: Comparativo, entre tempo de síntese, posicionamento e Roteamento, Área Ocupada, e Frequência de operação. 69
Tabela 6.1: Tamanho dos componentes na síntese da arquitetura. 87
Tabela 6.2: Custo da arquitetura em uma Virtex-6, apenas sintetizada e com place & route... 88
Tabela 6.3: Número de arestas dos grafos testados..... 89
Tabela 6.4: Tamanho do período dos grafos testados. 89
Tabela 6.5: tamanho do transiente dos grafos testados. 89
Tabela 6.6: Tempo de execução dos grafos na versão sequencial, em milissegundos. 90

Tabela 6.7: Tempo de execução dos grafos no FPGA, em milissegundos. 91

RESUMO

VENDRAMINI, Julio Cesar Goldner, M.Sc., Universidade Federal de Viçosa, Março de 2012. **Rede Omega Virtual em FPGA com reconfiguração em tempo de execução. Estudo de caso: Cálculo de Atratores em Redes Reguladoras de Genes**. Orientador: Ricardo dos Santos Ferreira.

As redes de interconexão multiestágio começaram a ser usadas na década de 50 em telefonia e continuam a ser usadas em muitas aplicações paralelas. Abordamos neste trabalho um estudo sobre as redes de interconexão Omega em FPGAs para desenvolvimento de arquiteturas paralelas e reconfiguráveis. Utilizando-as como uma camada virtual de reconfiguração acima da programação do FPGA. Analisamos o comportamento das redes e sua complexidade em área e latência. Desenvolvemos dois algoritmos de roteamento em hardware para o roteamento dessas redes, um utilizando memórias e outro utilizando codificador de prioridade. Mostramos também que o uso das redes Omega é viável em arquiteturas reconfiguráveis mapeadas em FPGA. Duas aplicações reais foram avaliadas. A primeira possui uma rede de interconexão global para interligar unidades de processamento heterogêneas em uma arquitetura reconfigurável de grão grosso. A segunda usa uma rede de interconexão em uma aplicação de bioinformática para interligar unidades. Esta aplicação explora um espaço de solução exponencial no cálculo de período de atratores. Os resultados obtidos mostraram um ganho de 2 a 3 ordens de grandeza em relação a solução com processadores de uso geral.

ABSTRACT

VENDRAMINI, Julio Cesar Goldner, M.Sc**.,** Universidade Federal de Viçosa, March, 2012. **Runtime Reconfiguration on Virtual Omega Networks. Case study: Attractors in models of Gene Regulatory Networks.** Adviser: Ricardo dos Santos Ferreira.

The multistage interconnection networks emerged in 1950 to be used on telephony systems. During the last decades, multistage networks have still been used on many parallel applications. This work analyzes Omega interconnection networks as a communication approach to be used on parallel architectures mapped on FPGAs. The proposed approach is based on a virtual layer above the FPGA's programmation as a logical level. Several topics were covered. First, the behavior of networks has been analyzed as a function of its workload. We show that Omega network is non-blocking under a partial workload. Moreover, we propose to use the radix4 switches which have a better routing capacity. In addition, the radix4 network reduces the area and the latency of an FPGA implementation. Besides, the regularity of the network was captured with FPGA synthesis tools. Third, one of the problems of multistage networks is the time to configure or routing the connections. We present two routing algorithms on hardware that can be used at runtime. One of the algorithms is based on embedded memories that has a low implementation cost. However, the algorithm spends between two and ten cycles to route one connection. The second algorithm uses a priority encoder and has a high cost compared to the first. Although, this algorithm only takes two cycles to route any connection. Finally, two case study are presented to show how a Omega networks is an efficient option as communication mechanism in FPGA based reconfigurable architectures. The first has a global interconnection network to interconnect heterogeneous processing units in a coarse-grained reconfigurable architecture (CGRA). The second uses an interconnection network in a bioinformatics algorithm. The algorithm computes the attractors on gene regulatory networks modeled with Boolean graphs and scale free topology. This algorithm explores an exponential space solution for the calculation of period attractors. The results showed a gain of 2-3 orders of magnitude over the solution implemented in software with general purpose processors.

1. Introdução

Nos últimos anos, os circuitos computacionais se tornaram maiores e mais complexos, isto foi possível devido ao avanço das técnicas de fabricação, que vem permitindo que circuitos possam ocupar áreas menores. A implementação de circuitos não é mais um desafio [Borkar, 2006], porém projetar circuitos grandes é cada vez mais complexo. A computação espacial é uma estratégia para melhorar o processamento e ao mesmo tempo diminuir a complexidade de desenvolvimento, onde o preço a ser pago para o aumento de desempenho é o aumento de área [Dehon, 2002].

A computação espacial replica unidades de processamento para distribuir o processamento entre elas, assim cria-se uma computação paralela, onde se faz necessário distribuir os dados de forma eficiente entre as unidades. Em geral é mais fácil integrar várias unidades simples de processamento do que desenvolver uma unidade complexa com o mesmo desempenho.

É cada vez maior o número de aplicações que demandam uma maior capacidade de processamento, onde existe um grande luxo de dados e uma demanda para distribuir os dados. As arquiteturas paralelas são vantajosas se comparado as soluções sequenciais neste contexto.

Outro problema é o tempo para projeto e implementação de um sistema. Para minimizar esse problema, o hardware programável pode ser usado como uma alternativa viável para desenvolvimento[Friedman, 2009][Coole, 2010][Mei, 2003]. Já que com o hardware programável é possível desenvolver arquiteturas com redução de custo, no tempo de testes. Tornando as arquiteturas programáveis ideais para sistemas de prototipação e também como uso comercial. Atualmente, o auge do hardware programável são os FPGAs. Os FPGAs são arquiteturas de grão fino que podem implementar qualquer circuito, limitado apenas pelo tamanho físico dos seus chips. Esta limitação tem diminuído a cada nova versão lançada de chips. Outro ponto importante é a potência e energia dissipada pelas arquiteturas. As soluções paralelas em hardware programável são uma alternativa que oferece uma relação número operações/energia mais atraente que as soluções paralelas baseadas em processadores de uso geral [Liu, 2008].

Um dos pontos mais importantes da computação paralela é a distribuição e comunicação entre as unidades de processamentos. Quanto maior a arquitetura, mais complexos são suas

interconexões e mais difíceis de posicionar os recursos. Atualmente, o mapeamento das aplicações em FPGA é complexo. Os algoritmos de posicionamento e roteamento são NP-completos [Mei, 2003] [Yoon, 2009]. Para minimizar o problema foram propostas várias soluções paralelas como as Arquiteturas Reconfiguráveis de Grão Grosso (CGRA - *Coarse-Grained Reconfigurable Architecture*)[Ferreira, 2010a]. Estas arquiteturas possuem a capacidade de configuração em nível de palavras, diferente do FPGA onde a configuração é em nível de bits, o que torna mais simples o mapeamento.

Uma qualidade fundamental em uma arquitetura paralela é que a mesma consiga executar tarefas distintas sem sofrer penalidades. Para isso, é interessante que a arquitetura se adapte de acordo com a necessidade da aplicação e consiga se reconfigurar para se adequar a tarefa que será executada. Ou seja, possua uma capacidade de adaptação dinâmica. Este trabalho propõe uma camada virtual que funcione acima da capacidade de programação dos FPGAs para criar arquiteturas reconfiguráveis flexíveis e com mapeamento simples para permitir a reconfiguração em tempo de execução.

Diferente da solução de reconfiguração parcial proposta pela Xilinx, onde se utilizam ferramentas para reconfiguração a nível físico reprogramando parcialmente o FPGA, a reconfiguração dinâmica aqui proposta, utiliza uma camada virtual que é sintetizada no FPGA e reconfigurada através de uma memória de configuração. Além disso, circuitos implementam em tempo de execução algoritmos mapeamento. Para reconfigurar a arquitetura de grão-grosso que funciona sobre a camada virtual do FPGA, utiliza-se a reconfiguração em nível de palavras, a sua configuração pode ser calculada em tempo de execução.

1.1 Motivação

Para uma arquitetura paralela ter uma capacidade de processamento eficaz, é necessário que a intercomunicação dos elementos seja eficiente. Ao mesmo tempo, as interconexões não podem ocupar um espaço muito grande dentro da arquitetura.

Este trabalho propõe o uso de redes de interconexão multiestágio em arquiteturas paralelas sobre uma camada virtual em FPGAs. Os objetivos são a redução do custo em tamanho e em latência, através da exploração da capacidade de roteamento das redes multiestágio em arquiteturas reconfiguráveis de grão grosso.

Trabalhos correlatos mostraram que arquiteturas com capacidades de reconfiguração parcial nos FPGAs são viáveis e permitem utilizar chips menores além de diminuir o consumo de energia [Hübner, 2006]. Além disto, as camadas virtuais em FPGA aumentam o desempenho, na ordem de centenas de vezes, do mapeamento e posicionamento da arquitetura. [Coole, 2010].

Todos os trabalhos aqui desenvolvidos surgiram a partir de uma demanda de alto desempenho para um problema de bioinformática abordado no Capítulo 6. A escolha de usar FPGA se deu, pois, trabalhos correlatos mostram que a utilização de FPGAs para resolver problemas de bioinformática [Lloyd, 2011] [Xia, 2011], podem ser acelerados centenas de vezes se comparadas a soluções sequenciais.

1.2 Proposta

A proposta desde trabalho é a implementação das redes multiestágio em FPGA e a verificação do comportamento em área e latência. As redes irão formar uma camada virtual reconfigurável acima do FPGA. Dando continuidade a trabalhos anteriores [Vendramini, 2010a][Ferreira, 2009].

Como contribuições, este trabalho propõe:

- Um estudo detalhado da área e latência das redes multiestágio Omega nos FPGAs.
- Desenvolvimento de algoritmos de roteamento para as redes Omega executados em hardware. Para poderem ser utilizados *Just-in-time*.
- Criação de uma camada reconfigurável virtual para interconectar unidades de processamento em tempo de execução.
- Duas aplicações onde a camada virtual é utilizada para intercomunicar elementos de duas arquiteturas paralelas distintas. A primeira aplicação é uma arquitetura reconfigurável de grão-grosso como rede de interconexão global interligando seus elementos de processamento. A segunda aplicação é para uso em aplicações de bioinformática para cálculo de atratores de redes reguladoras de genes.

1.3 Estrutura do Trabalho

O Capítulo 2 apresenta um levantamento da literatura sobre as redes de interconexão, abordando algumas pesquisas já realizadas, principalmente no uso de computação paralela. A

classificação das redes multiestágio e o contexto são apresentados, em especial para as redes bloqueantes, rearranjáveis e não bloqueantes. Para o contexto da dissertação, uma descrição dos FPGAs, suas vantagens e suas características também são apresentadas.

O Capítulo 3 apresenta um estudo sobre a rede multiestágio Omega. Neste estudo verificou-se que sua área e latência é O(N LgN) e O(Lg N) para implementações em FPGAs. Redes Omegas de vários tamanhos e características distintas foram comparadas. Analisou-se o impacto dos estágios extras na capacidade de roteamento.

O Capítulo 4 apresenta o estudo do roteamento das redes Omega, analisando a capacidade de roteamento com várias cargas de trabalho, com e sem estágios extras. Analisamos também a capacidade de roteamento na presença de *multicast*. Analisamos redes de vários tamanhos e duas opões de comutadores internos. Apresentamos também dois algoritmos em hardware para o roteamento da rede. Um utilizando memória embarcada e outro utilizando um codificador de prioridade parcial.

O Capítulo 5 propõe uma arquitetura reconfigurável de grão grosso utilizando redes Omega como uma interconexão global de unidades de processamento heterogêneas. Neste capítulo é mostrado o funcionamento da arquitetura, além dos algoritmos necessários para fazer o mapeamento dos grafos de fluxo de dados. A arquitetura funciona sobre uma camada virtual implementada em um FPGA.

O Capítulo 6 descreve uma arquitetura para uso em bioinformática na qual temos também uma rede Omega interligando as unidades de processamento. Um algoritmo paralelo para o cálculo dos períodos de redes reguladores é apresentado. Esses períodos são conhecidos como atratores e tem importância para verificar o processo de evolução das células na biologia. Essa arquitetura funciona sobre uma camada virtual e foi implementada em um FPGA. A comunicação com externa foi interfaceada com o processador softcore Microblaze da Xilinx.

2. Revisão Bibliográfica

A “computação paralela” sempre foi e continua sendo um grande desafio. As redes de interconexão tem um papel fundamental na implementação de soluções paralelas. Porém, criar novas arquiteturas paralelas em VLSI demanda tempo de desenvolvimento. Os FPGAs são uma alternativa na qual o desenvolvimento pode ser feito com mais flexibilidade e facilidade, reduzindo a demanda no tempo de teste e verificação. O único problema do FPGA é que por ser uma arquitetura com alta flexibilidade, possui uma elevada complexidade de síntese, mapeamento e roteamento de arquiteturas complexas.

Neste trabalho analisamos redes de interconexão multiestágio em arquiteturas paralelas em FPGA. As redes multiestágios já foram usadas em sistemas de emulação e prototipação baseados em Multi-FPGAs [Barbie, 1999], eram necessários vários chips de FPGAs e redes de interconexão para emular um sistema. Ao invés de usar chips *crossbar*, que tem custo $O(n^2)$, alguns trabalhos propuseram dedicar um ou mais FPGAs à interconexões, aumentando a flexibilidade. Este trabalho difere dos anteriores ao propor o uso das redes multiestágios internamente no FPGA em conjunto com elementos de processamento e memória na implementação eficiente de arquiteturas paralelas.

2.1 Redes de Interconexão Multiestágio

2.1.1 Classificação das Redes Multiestágios

Uma rede de interconexão pode ser descrita matematicamente como um conjunto de conexões ou permutações de entrada/saída que ela é capaz de realizar ao mesmo tempo. As redes *crossbar* são um exemplo de rede capaz de realizar qualquer permutação de entrada/saída. Entretanto seu custo em comutadores é $O(N^2)$ para N entradas/saídas. As redes multiestágios só podem realizar um subconjunto das permutações completas. A vantagem é a redução do número de comutadores para O(N lg N).

Uma rede multiestágio com N entradas e N saídas é formada por uma série de colunas de comutadores (estágios), onde os comutadores do estágio j se conectam aos comutadores do

estágio j+1. Formalmente a rede é definida pela tupla M(N,E,R,P,B), onde N é o número de entradas e saídas, E é o número de estágios, R é o radix do comutador, P é o padrão de permutação de bits entre os estágios e B é o número de configurações para cada comutador.

Como exemplo, suponha a Tupla M(64, 6, 2, Rot. 1 bit pra esquerda, 4). Teríamos então uma rede de tamanho 64, com 64 entradas e 64 saídas. Os comutadores são radix2 com duas entradas e duas saídas. Cada comutador pode realizar 4 conexões diferentes, 0->0 e 0->1, ou 0->0 e 1->1, ou 0->1 e 1->0, ou 1->0 e 1->1. Teremos então (Lg_2 64) ou 6 colunas de comutadores. O padrão de comunicação é rotacionar 1 bit para a esquerda. Ou seja, a entrada 4(000100) ligaria na entrada 8(001000), a entrada 35(100011) ligaria na entrada 7(000111).

As redes de interconexão multiestágio podem ser classificadas como: bloqueantes, rearranjáveis e não bloqueantes, extraído de [Grammatikakis, 2000]. A Figura 2.1 ilustra uma classificação mais ampla das redes de interconexão.

Figura 2.1: Classificação das Redes de interconexão multiestágio. [Grammatikakis, 2000]

Antes de detalhar a classificação da rede em função da sua capacidade de roteamento, iremos introduzir o conceito de conexão ou permutação completa da rede que equivale a uma

permutação das entradas como ilustra a Figura 2.2 onde as entradas i_0 à i_7 se conectam as saídas o_3,o_2,o_5,o_0,o_7,o_6,o_4 e o_1, respectivamente. Ou seja, i_0 conecta à o_3, i_1 conecta à o_2, i_2 conecta à o_5, i_3 conecta à o_0, i_4 conecta à o_7, i_5 conecta à o_6, i_6 conecta à o_4, i_7 conecta à o_1.

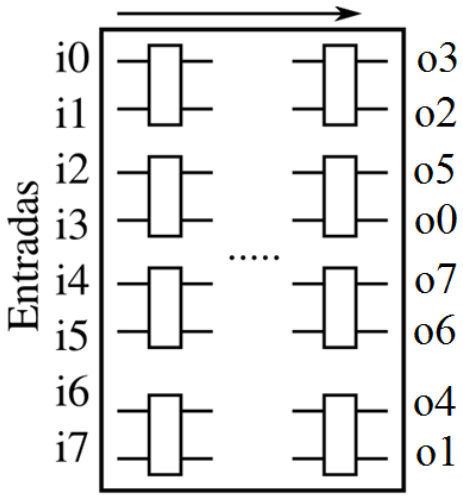


Figura 2.2: Permutação de entradas.

Uma rede é bloqueante quando não consegue rotear todas as permutações de entrada/saída. A rede Omega ou *Butterfly* são exemplos de redes bloqueantes. A Figura 2.3 ilustra a rede Omega [Lawrie, 1975] com 16 entradas/saídas que é bloqueante. Se considerarmos comutadores 2x2, a rede Omega terá Lg_2 N estágios, no exemplo da Figura 2.3 tem quatro estágios. Para facilitar a nomenclatura, os termos logaritmos são da base 2. O número de comutadores é N/2*Lg N e o número de comutadores que um sinal tem que atravessar entre a entrada e a saída da rede é Lg N, ou seja, a latência é O(Lg N) e o custo em comutadores O(N Lg N). É determinado pelo padrão de conexão entre os estágios o tipo de rede. Estas redes são conhecidas como redes de permutação de bits. No caso da rede *Omega*, o padrão de bit é uma rotação à esquerda no endereço da linha, também conhecido como *perfect-shuffle*. Por exemplo, a linha 1 ou 000**1** em binário irá conectar a linha 2 ou 00**1**0 em binário como ilustrado na Figura 2.3 ou seja a linha $x_3x_2x_1x_0$ conecta a linha $x_2x_1x_0x_3$ para 16 entradas. Depois de realizada a rotação, o sinal passa pelo comutador do estágio, que pode trocar o último bit, por isso a rede *Omega* também é conhecida como *Shuffle-Exchange*. Apesar de bloqueante, a rede *Omega* realiza vários padrões como conectar a entrada i_j à saída o_{j+t} onde t é um inteiro e representa uma translação da entrada, outros padrões são detalhados em [Lawrie, 1975]. A *Butterfly* conecta a linha $x_{n-1}...x_i...x_1x_0$ na linha $x_{n-1}...x_0...x_1x_i$ ou seja troca o bit x_i pelo x_0 em função do estágio da rede. Tanto a *Omega* quando a *Butterfly* são conhecidas como redes de *Bayan* onde uma entrada pode alcançar qualquer saída, e existe apenas um único caminho

para rotear uma entrada em uma saída. Esta característica simplifica o roteamento que será detalhado no Capítulo 4. Vários padrões de redes bloqueantes foram propostos nos anos 70, e no inicio da década de 80 foi demonstrado que estes padrões são equivalentes [Wu, 1980].

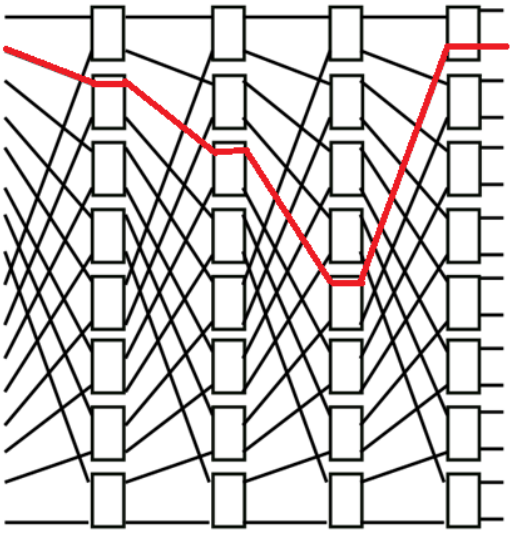


Figura 2.3: Rede Omega.

Uma rede é rearranjável se é possível realizar todas as permutações, desde que as conexões já estabelecidas possam ser rearranjadas para não gerar conflitos. A rede *Benes* [Benes, 1965] ilustrada na Figura 2.4 é um exemplo de rede rearranjável. Entretanto o custo da rede *Benes* é o dobro da rede *Omega*, tem pelo menos 2*(Lg N) - 1 estágios, o que aumenta também sua latência. Assim como as redes bloqueantes, as redes rearranjáveis são organizadas em classes [Yeh, 1992].

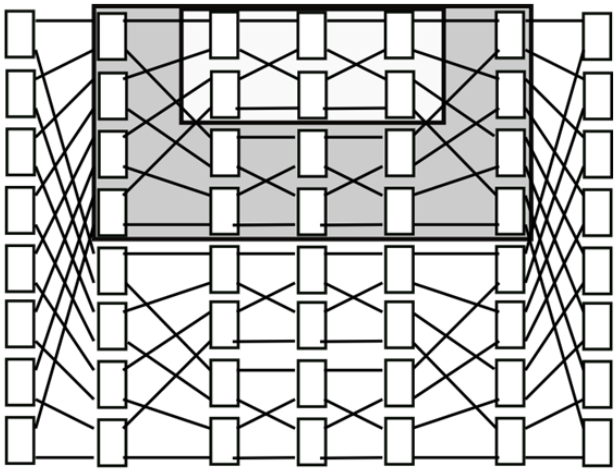


Figura 2.4: Rede rearranjável Benes

A rede é estritamente não bloqueante quando as requisições de conexão entre entrada e saída chegam de forma assíncrona e é sempre possível roteá-las. A rede *Clos* é um exemplo de rede não bloqueante. Entretanto, o custo dos comutadores aumenta significativamente, como ilustra a Figura 2.5 que apresenta uma rede *Clos* genérica com três estágios proposta em [Clos, 1953]. O primeiro estágio tem m comutadores n x k, o segundo estágio tem k comutadores m x m e o último estágio tem m comutadores k x n, para n entradas/saídas.

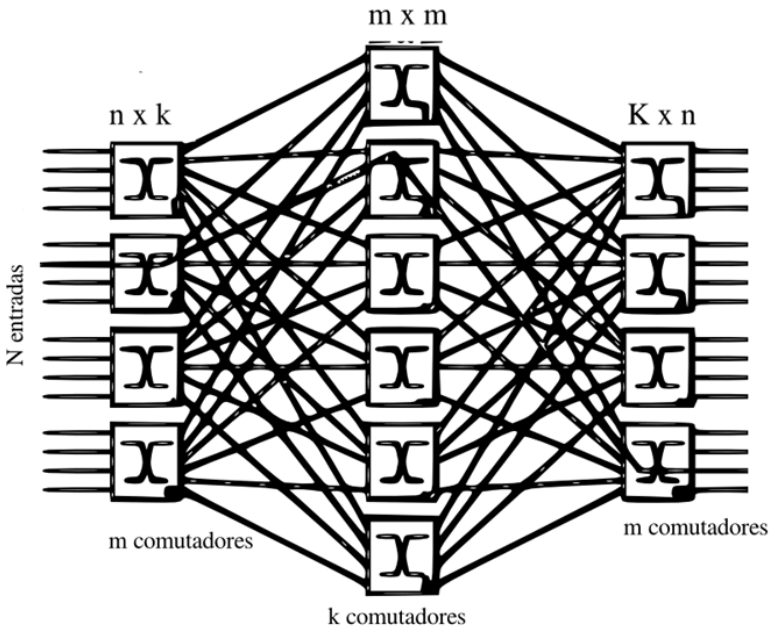


Figura 2.5: Rede não bloqueante Clos.

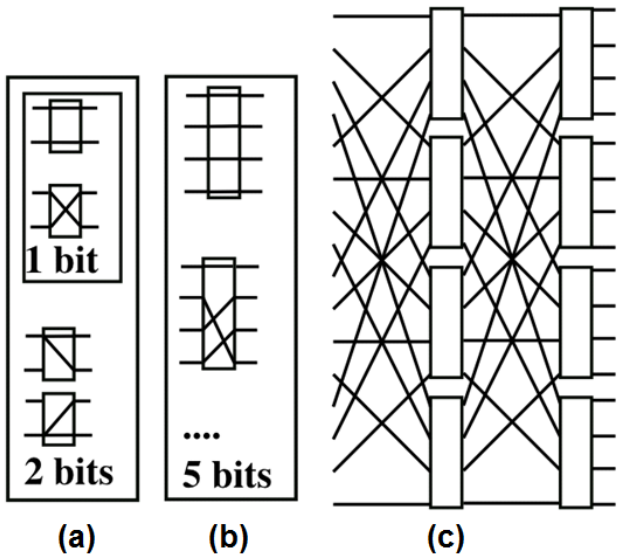


Figura 2.6: Comutadores: (a) 2x2; (b) 4x4 (c) Radix4.

Além da classe da rede, bloqueante, não bloqueante ou rearranjável, o custo e a capacidade de roteamento dependem da granularidade do comutador. O comutador da rede pode ser simples como um comutador 2x2 (veja Figura 2.6(a)) ou mais complexo com MxK entradas e saídas como os comutadores da rede Clos (veja Figura 2.5). Quanto maior o comutador, menor será o número de estágios. Ou seja, existe um custo benefício entre a complexidade do comutador e o número de estágios. Para comutadores 2x2 ou radix2, uma rede bloqueante terá Lg_2N estágios como vimos para a rede Omega na Figura 2.3. Agora, se usarmos comutadores 4x4 ou radix4, o número de estágios será Lg_4N como ilustrado na Figura 2.6(b) e na Figura 2.6(c) para uma rede Omega radix4 com 16 entradas que terá apenas 2 estágios.

Outro ponto importante é o número de bits de configuração para programar a rede em função do modo de operação. Um comutador simples 2x2 tem um bit de configuração quando opera no modo direto ou cruzado. Porém se dois bits de configuração forem usados pode-se fazer um *broadcast* local (veja Figura 2.6(a)). A configuração *broadcast* local serve para *broadcast* global (onde uma entrada é conectada a todas as saídas) ou *multicast* global (onde uma ou mais entradas são conectadas a subconjuntos de saídas). O uso de *multicast* dobra o número de bits para radix2.

Um comutador 4x4 terá 24 configurações simples sem multicast 1 para 1, ou seja, serão necessários no mínimo 5 bits. No caso de *multicast* ou *broadcast*, cada saída do comutador pode receber qualquer uma das 4 entradas. Portanto, são necessário 2 bits por saída e 8 bits para o comutador 4x4.

2.1.2 Trabalhos Correlatos

As redes multiestágios surgiram nas décadas de 50 e 60 para o sistema de telefonia [Clos, 1953][Benes, 1965]. Na década de 70, novas redes surgiram para computadores paralelos. Um exemplo é a rede Omega [Lawrie, 1975], cuja proposta inicial era a interconexão de processadores e memórias em máquinas paralelas. Alguns padrões de comunicação como os algoritmos de FFT e ordenação são realizados de forma eficiente nas redes Omega. Durante os anos 70 e 80 até o início dos anos 90, as pesquisas foram focadas nas classes de equivalência entre os diversos tipos de redes [Wu, 1980] [Yeh, 1992] e algoritmos paralelos de roteamento foram realizados [Yeh, 1992]. Outras redes equivalentes também sugiram como a *fat-tree* [Leiserson, 1985].

Durante os anos 80 e 90 foram feitos estudos e implementações físicas em VLSI para avaliar o potencial de implementação das redes multiestágios em silício [Dinizt, 1999][Dehon, 2000]. Apesar da complexidade O(N Lg N) em termos de número de comutadores, as implementações VLSI mostram um custo $O(N^2)$ em área de silício [Dinizt, 1999], e exigiam também multi-camadas de fios no nível físico. Para circuitos reconfiguráveis como FPGA, as redes também foram avaliadas como um mecanismo de interconexão programável. Em [Dehon, 2000], uma nova arquitetura física para FPGA com o uso de multiestágios para interligar as unidades lógicas ou *Lookup Table* (LUT) foi proposta. Ao invés da topologia bidimensional com segmentos de barramentos em *clusters* (tipo ilha), foi proposto usar interconexões multiestágio. Entretanto, a construção eficiente do circuito depende de recursos multi-camadas para os fios. Porém, nenhum FPGA comercial foi construído com redes multiestágios e continuam a ser fabricados no estilo ilha.

Com a popularização dos FPGA nos anos 90, alguns trabalhos propunham sistemas grandes compostos de vários chips FPGA e uma arquitetura de comunicação entre eles, conhecidas como arquiteturas multi-FPGA. Os FPGAs comerciais eram usados para realizar a computação, como por exemplo, a emulação de circuitos, e também para implementar interconexão. A ideia era uma rede de interconexão no nível lógico, sintetizando a multiestágio sobre o FPGA, como as redes Clos [Barbie,1999], Folded-Clos [Lin,1997], redes *crossbar* parciais [Ejnioui,1999]. Neste caso, o FPGA inteiro é usado para implementar uma rede e interligar os outros chips de FPGA que realizam computação.

Recentemente, trabalhos propuseram o uso de redes em arquiteturas reconfiguráveis de grão grosso [Tanigawa, 2008][Ferreira, 2008][Ferreira, 2009a] e sistemas com multiprocessadores MPSoCs [Neji, 2008]. Em [Tanigawa, 2008], uma arquitetura bidimensional com redes *Benes* a ser sintetizada em VLSI foi proposta, ou seja, o trabalho propõe construir fisicamente a rede em circuito para interligar unidades funcionais. Os resultados mostraram que o layout é bem compacto, cerca de 1.000 vezes menor que um processador multicore, pois as ligações foram implementadas no nível serial. Mesmo com um tempo de relógio quatro vezes maior, a versão multiestágio apresenta o mesmo tempo de execução para aplicações multimídia quando comparada aos processadores multicore. Porém o mapeamento das aplicações na arquitetura foi feito manualmente. Semelhante a abordagem anterior, porém acoplado a um processo de tradução binária, no qual em tempo de execução o código binário da aplicação é mapeado na arquitetura reconfigurável, o uso de redes multiestágio em VLSI foi proposto em [Ferreira,

2008]. Neste trabalho, as redes *Omega* [Lawrie, 1975] são usadas para interligar as linhas de um arranjo bidimensional de unidades funcionais acopladas a um processador MIPS. Um algoritmo de posicionamento e roteamento em hardware foi apresentado, mostrando que o roteamento pode ser realizado em tempo de execução. Este trabalho também demanda a construção física em VLSI de um circuito reconfigurável com as redes.

Posteriormente, as redes *Omega* foram implementadas em FPGAs no nível lógico [Ferreira, 2009b]. A idéia era melhorar a capacidade de roteamento de um *grid* de ALUs para grafos de fluxos de dados. Os grafos eram mapeados no *grid* que não possuía a capacidade de roteamento, apenas comunicação direta dos vizinhos (norte, sul, leste e oeste). Quando dois vértices do grafo de fluxo de dados não eram vizinhos na arquitetura, a comunicação deles era roteada através da multiestágio. Este trabalho mostrou dois resultados interessantes. Primeiro, as multiestágios, como recurso adicional de roteamento apresentaram uma área menor que a integração da capacidade de roteamento nas unidades do *grid*. Segundo, mesmo usando uma rede com conflitos (redes Omega), como a demanda de roteamento era da ordem de 30% dos sinais, a multiestágio realiza com sucesso o roteamento. Além disso, o tempo de execução do algoritmo de roteamento implementado em software, foi da ordem de milissegundos que possibilita sua integração em ambientes de compilação *Just-in-Time*. Foram usadas redes com largura de 32 bits para as palavras de dados.

Recentemente, um trabalho avalia a área e o desempenho dos FPGAs com as redes multiestágios em sistemas multiprocessadores do tipo Multiprocessors *System on Chip* (MPSoCs) [Neji,2008] no contexto de NoC (*Network on chip*). São sintetizados até 8 processadores miniMIPS, onde cada um possui uma memória de instrução dedicada e usam a rede para comunicar com 8 módulos de memória de dados. Os resultados mostram que o tempo de síntese e a área das multiestágios é bem menor quando comparada as redes *crossbar*. A complexidade da área ocupada pela rede foi de O(N Lg N) após a síntese. Porém não se diz qual a largura da palavra, os miniMIPs são em geral de 32 bits. O roteamento foi implementado por comutação de pacotes no nível de processadores, diferente do foco dessa dissertação onde o roteamento é realizado com comutação de circuitos para interligar unidades funcionais.

2.2 FPGA

Praticamente todos os chips que encontramos no dia-a-dia, como circuitos de televisões, aparelhos de DVD, celulares, etc, já possuem a programação de fábrica, ou seja, já vêm pré-programados, tendo definidas suas funcionalidades desde a fábrica. Existe também uma categoria menos comum, de hardware programável, que tem suas funcionalidades definidas exclusivamente pelos usuários, e não por fabricantes. Dentre uma variedade de hardware programável, podemos destacar os FPGAs, (*Field Programmable Gate Array*), ou Arranjo de Portas Programável em Campo.

O FPGA foi criado pela Xilinx Inc., e teve o seu lançamento no ano de 1985 como um dispositivo que poderia ser programado a nível físico de acordo com as necessidades do desenvolvedor. O FPGA é composto basicamente por três tipos de componentes: blocos lógicos configuráveis (CLB), chaves de interconexão (*Switch Matrix*) e blocos de entrada e saída (IOB). Os blocos lógicos são dispostos de forma bidimensional, as chaves de interconexão são dispostas em formas de trilhas verticais e horizontais entre as linhas e as colunas dos blocos lógicos, na Figura 2.7 temos uma noção visível da arquitetura do FPGA.

- CLB (*Configuration Logical Blocks*): Circuitos idênticos, construído pela reunião de flip-flops e a utilização de lógica combinacional, utilizando LUTs (*lookup tables*). Utilizando os CLBs, um usuário pode construir elementos funcionais lógicos. As LUTs, formam lógicas combinatórias de 4 ou 6 entradas e uma saída, que representam qualquer função de até 4 variáveis ou 6 variáveis de um bit, respectivamente. As LUTs armazenam a tabela-verdade das funções que representam.
- *Switch Matrix* (chaves de interconexões): São as trilhas que fazem a conexões de todos os CLBs e IOBs, ela são em forma de ilha e possuem ligações tanto na vertical quanto na horizontal. Ela tem capacidade de interligar os componentes de formas variadas. Geralmente, a configuração é estabelecida por programação interna das células de memória estática, que determinam funções lógicas e conexões internas implementadas no FPGA entre os CLBs e os IOBs. O processo de escolha das interconexões é chamado de roteamento.
- IOB (*Input/Output Block*): São circuitos responsáveis por criar uma interface entre as entradas/saídas físicas do FPGA e as provenientes das entradas/saídas das

combinações de CLBs. São basicamente *buffers*, que funcionarão como um pino bidirecional entrada e saída do FPGA.

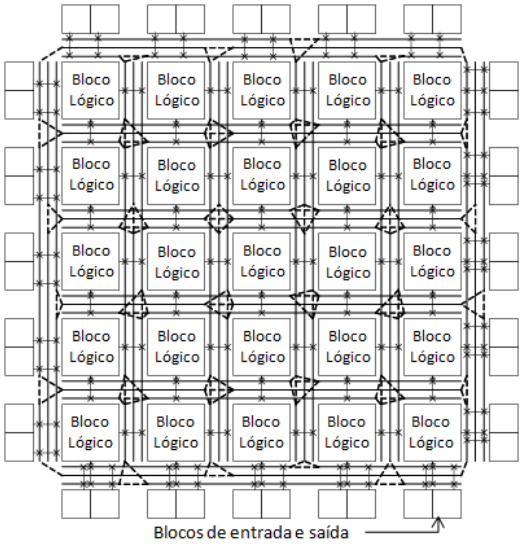


Figura 2.7: Arquitetura interna do FPGA

Devido à facilidade de desenvolvimento comparada ao VLSI, os FPGAs vêm sendo muito utilizados para prototipação de circuitos e testes, implementações de algoritmos em hardware e como aceleradores para problemas de alto custo de processamento. Devido à regularidade do circuito, a tecnologia atual permite a integração de milhões de blocos lógicos em um único FPGA. Além disso, FPGAs recentes são heterogêneos e incorporam circuitos com multiplicadores e memória, que são úteis para diversas aplicações como robótica, bioinformática, sistemas embarcados, multimídia, telecomunicações dentre outras [Hauck, 2007].

Os FPGAs possuem um alto grau de paralelismo de grão fino podendo se ajustar ao problema. Para problemas regulares como processamento de imagens, existem ferramentas para mapeamento da computação, porém para problemas gerais, foram e continuam sendo realizadas pesquisas nas últimas décadas para o desenvolvimento de ferramentas de síntese de alto nível.

Como já mencionado, o FPGA é um hardware programável, a arquitetura é reconfigurável em tempo de execução. Para configurar é necessário sintetizar, mapear e rotear a arquitetura desejada. O FPGA tem também a capacidade de reconfiguração parcial, onde se configura apenas uma parte dele, mantendo as demais partes com a configuração antiga.

O FPGA permite também que criemos uma camada de reconfiguração virtual, ou seja, é uma arquitetura implementada diretamente no FPGA que possui um padrão de configuração diferente da que o FPGA possui e pode ser reconfigurada em tempo de execução bastando ter sua configuração armazenada em alguma memória no FPGA. Sem precisar passar por todo o processo de síntese, mapeamento e roteamento que uma arquitetura padrão para FPGA necessita.

Atualmente existem no mercado chips FPGA como o Virtex-7 com mais de 6 bilhões de transistores e 2 milhões de blocos lógicos utilizando tecnologia de 28nm.[Xilinx, 2012]. Podemos ver na Figura 2.8(a) um FPGA, e na Figura 2.8(b) a placa de protótipo da Xilinx com uma Virtex-6 utilizada no desenvolvimento deste projeto.

Figura 2.8: (a) FPGA Virtex-6; (b) placa de protótipo FPGA, Xilinx ML605, utilizada no projeto.

3. Rede Multiestágio Omega

As redes multiestágio podem ser classificadas em bloqueantes, rearranjáveis, e não bloqueantes. No presente trabalho utilizamos a rede Omega dando continuidade a trabalhos anteriores [Vendramini, 2010] que mostram um bom custo benefício entre a capacidade de roteamento e o custo da implementação em FPGA.

Neste capítulo são apresentados os resultados de um estudo detalhado da implementação das redes Omega em FPGA [Vendramini, 2010a] A implementação foi parametrizada para estudar o custo em área em função do número de entradas/saídas, do número de estágios extras, da largura da palavra, do número de bits de configuração e da capacidade ou radix do comutador. O atraso com e sem registros entre os estágios também foi avaliado.

A rede Omega é bloqueante. O padrão de ligação igual em todos seus estágios. O número de comutadores é O(N LgN), e espera-se que o custo em recursos no FPGA possua a mesma complexidade. A latência que é O(LgN). Uma vantagem de utilizar essa rede bloqueante é a redução do custo em área e latência se comparada a uma rede rearranjável ou a uma não-bloqueante. Uma desvantagem é a redução da capacidade de roteamento. Entretanto com iremos apresentar nos capítulos 4, 5 e 6 a rede Omega é adequada para várias aplicações.

A rede Omega com LgN estágios possui apenas um caminho entre uma determinada entrada A e uma determinada saída B. Entretanto, se adicionarmos estágios extras a rede, o número de caminhos da entrada A para a saída B dobra a cada estágio adicionado. Cada estágio extra aumenta a rede em uma coluna de comutadores. Para um estágio extra, temos dois caminhos distintos, para dois estágios extras teremos quatro caminhos distintos.

3.1 Metodologia para os testes

Os testes foram executados em ambiente Microsoft Windows, utilizando a ferramenta de desenvolvimento da Xilinx, ISE 11.4, utilizando código parametrizado em VHDL em função do radix dos comutadores, número de estágios, bits de configuração, largura da palavra, etc. O objetivo foi avaliar o custo das redes em área ocupada, em latência e como as ferramentas de síntese de FPGA capturam uma descrição em alto nível das redes. Para a arquitetura alvo usamos uma Virtex6 XC6LVX240T que possui 150.720 LUTs de 6 entradas, 301.440 flip-flops,

416 módulos de memória e 768 DSPs (unidades de processamento digital embarcadas). Este dispositivo foi selecionado devido ao fato da disponibilidade física de uma placa com este circuito no Departamento de Informática da Universidade Federal Viçosa que foi adquirido com recursos do Edital Universal da FAPEMIG [FAPEMIG, 2011].

As memórias embarcadas são chips de memórias embutidos no FPGA, com largura de 36 bits e 512 linhas, elas podem ser utilizadas para armazenar qualquer tipo de valor e podem ser agrupadas para formar memórias maiores. As unidades de processamento digital embarcadas(DSPs) também estão embutidos no FPGA distribuídos em vários locais para facilitar e diminuir o tamanho de arquiteturas que precisam de multiplicadores. Eles podem realizar várias operações aritméticas que diminuem o consumo em área e atraso no FPGA, dentre elas podemos citar: multiplicadores, somadores, deslocamento de bits.

3.2 Resultados Experimentais

Nos experimentos, dois tipos de comutadores foram usados: Radix2 e Radix4. O número de entradas/saídas da rede é uma potência do radix do comutador para redes homogêneas. Ou seja, para radix2 tem-se 2^N entradas e 4^N para radix4. O número de comutadores utilizados é $N*Lg_{(radix)}N$ e o número de colunas é $Lg_{(radix)}N$. A Figura 3.1 ilustra os dois comutadores. Um comutador com radix-N pode ser visto com uma pequena rede *crossbar* com N entradas/saídas. Se aumentar o tamanho do radix for igual ao número de entradas da rede, teremos uma rede *crossbar* que tem custo $O(n^2)$.

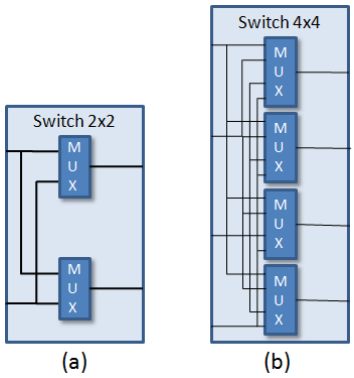


Figura 3.1: (a) comutador em rede radix2; (b) comutador em rede radix4.

3.2.1 Área

No primeiro experimento foi avaliada a evolução da área ocupada em recursos (LUTs) em função do número de entradas e da largura da palavra. A Tabela 3.1 apresenta os resultados para redes Omega radix2 com o número de entradas variando de 32 a 512 e com a largura da palavra em bits variando de 1, 8, 16 e 32 bits. A Figura 3.2 ilustra uma rede Omega 8x8 com largura de 16 bits, com 16 bits, todos as ligações da rede são barramento de 16 bits de largura.

Área ocupada em LUTs					
	Número de entradas				
Largura	**N = 32 (LUTs)**	**N = 64 (LUTs)**	**N = 128 (LUTs)**	**N = 256 (LUTs)**	**N = 512 (LUTs)**
1 bit	392	425	1108	1838	4017
8 bits	1227	2443	6039	12592	29107
16 bits	2251	4747	11501	24880	57779
32 bits	4299	9355	22765	49456	115123

Tabela 3.1: Tamanho da Rede Multiestágio Omega. Sintetizada em uma Virtex6 XC6VLX240T com 150730 Luts.

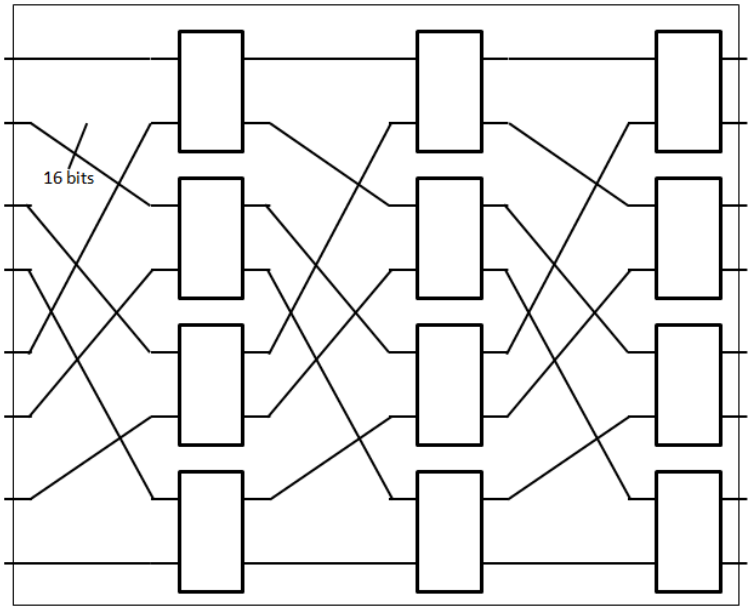


Figura 3.2: Rede Omega 8x8 com largura de 16 bits.

Podemos observar na Tabela 3.1, que o espaço ocupado no FPGA para a rede Omega (radix2) em função do tamanho da entrada cresce com a complexidade O(N LgN), ou seja, a ferramenta de síntese captura a regularidade da rede. Para redes pequenas como 32 entradas e largura de 1 bit, tem-se um custo inicial. À medida que o tamanho da rede aumenta, o custo cresce com uma taxa menor que N Lg N. Por exemplo, a rede quadruplica de 32 entradas para 128 entradas, mas o tamanho cresce menos que 3 vezes. O correto seria crescer mais de 4 vezes.

Já de 256 entradas para 512 entradas, a rede cresce 4017 LUTs/1838 LUTs = 2,18 que segue o tamanho N Lg N = (512*Lg(512))/256*(Lg 256) = 2,27. Em termos de espaço, vemos que uma rede de 512 entradas com largura de 32 bits ocupa 70% do FPGA. Entretanto, a complexidade de um sistema com 512 conexões de 32 bits é bem elevada. Para o FPGA Virtex6 avaliado, redes com o tamanho máximo de 128 entradas e 32 bits são recomendadas, pois a rede ocupará 15% da área do FPGA, sendo que o restante pode ser usado para implementação de unidades de processamento.

Outro ponto é que ao aumentar de 1 bit para 8 bits, a área em LUTs aumentou em média apenas 5,7 vezes, ao aumentar para 16 bits em relação a 1 bit, a área aumentou em 11 vezes em média, e para 32 bits o aumento médio foi de 22 vezes. Mostrando que a regularidade da rede é um aspecto importante capturado pela síntese. Além disso, as redes com largura pequena em bits apresentam um custo proporcionalmente maior.

3.2.2 Latência

No primeiro experimento também foi avaliada a latência da rede em função do número de entradas e largura da palavra. A Tabela 3.2 mostra os resultados da latência para as redes da Tabela 3.1 utilizando a ferramenta de síntese da Xilinx. Podemos observar que a latência não varia com a largura de bits, apenas varia o número de estágios. A cada coluna estamos acrescentando um estágio a mais na rede, ao dobrar o seu tamanho uma coluna a mais é acrescentada. Uma rede com 8 entradas radix2 terá 3 estágios e 12 comutadores. Se aumentar para 16 entradas, o número de comutadores passa para 32, porém apenas um estágio é adicionado, como podemos observar na Figura 3.3. Pode-se observar na Tabela 3.2, que a rede de 32x32, que tem 5 estágios, tem a latência de 1,97 ns. Ao dobrar para uma rede de 64x64 teremos 6 estágios e a latência aumentou para 2,28, ou seja, 0,33 ns para atravessar um estágio a mais. Podemos observar que a latência aumenta em média 0,34 ns por estágio ao crescer de 32 para 64 até 512. Ou seja, a rede tem um comportamento linear em atraso.

Se adicionarmos um estágio, a latência aumenta 0,34 ns. Este recurso pode ser usado para aumentar a capacidade de roteamento da rede com estágios extras. Cada estágio extra dobra o número de caminhos para um par de entrada e saída. Entretanto, vale a pena ressaltar que os caminhos múltiplos são compartilhados, dobrar o número de caminhos entre A e B não implica em dobrar a capacidade total da rede. Ademais, uma rede bloqueante como a rede

Omega irá saturar em um máximo de 2 Lg N estágios. Mais estágios do que Lg N não alteram a capacidade de roteamento.

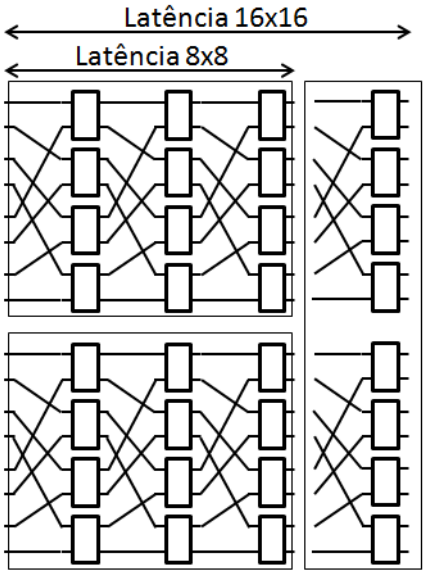


Figura 3.3: Rede Omega 16x16, comparativo com rede Omega 8x8.

Latência da rede multiestágio Omega (ns)					
	Número de entradas				
Largura	**32**	**64**	**128**	**256**	**512**
1 bit	1,97	2,28	2,64	2,95	3,32
8 bits	1,97	2,28	2,64	2,95	3,32
16 bits	1,97	2,28	2,64	2,95	3,32
32 bits	1,97	2,28	2,64	2,95	3,32

Tabela 3.2: Latência (ns) Rede Multiestágio Omega. Sintetizada em uma Virtex6 XC6VLX240T.

3.2.3 Bits de Configuração

Outro ponto importante é o número de bits para armazenar a configuração da rede. Os bits podem ser armazenados em módulos de memória embarcados ou em LUT na família de FPGA Virtex6. No segundo experimento foi avaliado o número de LUTs e módulos de memória embarcados necessários para o número de entradas variando de 32 a 512. Uma rede com N entradas e radix2 terá (N/2)*LgN comutadores. Cada comutador tem 1 ou 2 bits de configuração em função do modo *unicast* ou *multicast*, respectivamente como exemplifica a Figura 3.4. Por exemplo, uma rede com 64 entradas 64/2 * Lg 64 = 32*6 comutadores. Cada módulo de memória tem a largura de 36 bits. Ou seja, são necessários apenas 6 módulos para armazenar a configuração, a Figura 3.6 ilustra uma rede de tamanho 64x64.

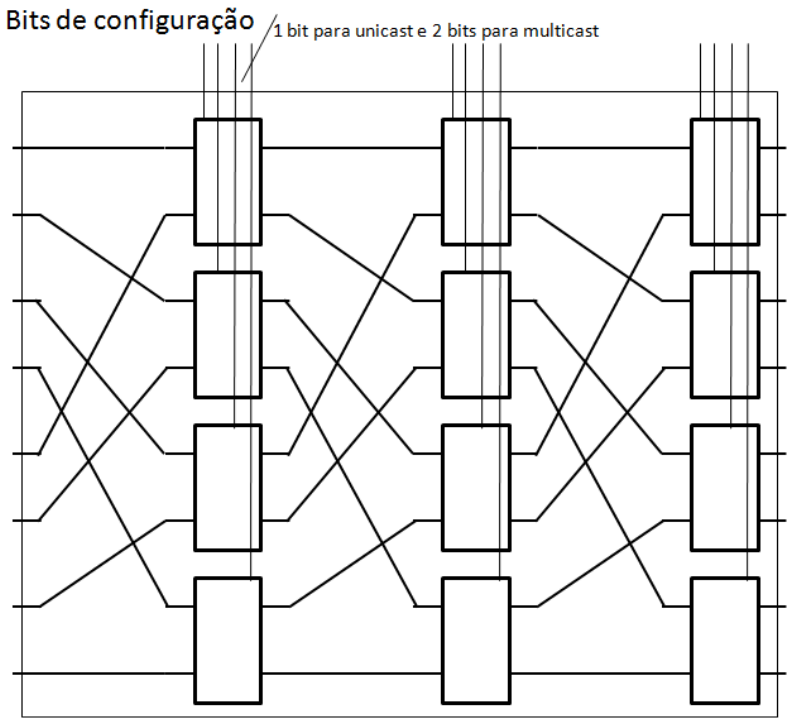


Figura 3.4: Rede Omega com as entradas dos bits de configuração.

A Tabela 3.3 apresenta os resultados do número de memórias embarcadas necessárias para armazenar as configurações das redes. Os resultados de área ocupada da rede e a taxa de crescimento não foram alterados e são os mesmos da Tabela 3.1. Uma Virtex6 tem 416 módulos, para a maior configuração avaliada, na de tamanho 512 são usados 72, ou seja, apenas 17% dos módulos. A primeira coluna com redes com 32 entradas usa LUTs para armazenar a configuração, enquanto que as outras colunas com redes de 64 à 512 entradas, a configuração é armazenada nas memórias embarcadas (BRAMs), visto na Figura 3.5. A ferramenta da Xilinx faz a escolha pelas LUTs para reduzir o uso dos recursos do FPGA na rede de 32, porém para as outras possibilidades, a ferramenta verifica que os módulos são mais adequados para armazenar as configurações.

Na arquitetura proposta na Figura 3.5, a configuração inteira da rede é armazenada em uma única linha. Como cada memória pode armazenar até 512 linhas, o conjunto de M módulos irá ter 512 linhas. Ou seja, sem aumentar o custo da implementação, podemos armazenar 512 configurações diferentes de interconexão e programá-las no FPGA estaticamente ou dinamicamente. Uma computação que envolva até 512 padrões diferentes de comunicação entre seus elementos pode ser reconfigurada nesta arquitetura. No caso de uma rede com 128 entradas, usamos apenas 14 módulos para armazenar os bits de configuração e 17% das LUTs do FPGA para implementar a rede, sendo que praticamente todos os módulos de memória

(mais de 400) juntamente com 80% do FPGA poderão implementar recursos de computação. A Figura 3.5 ilustra esta arquitetura abordada.

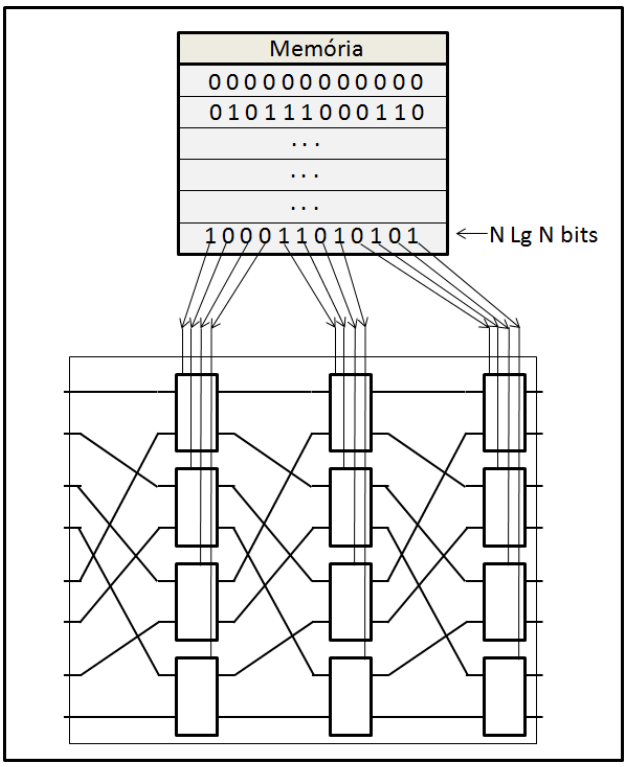


Figura 3.5: Rede Omega utilizando Memórias para armazenar suas configurações de roteamento.

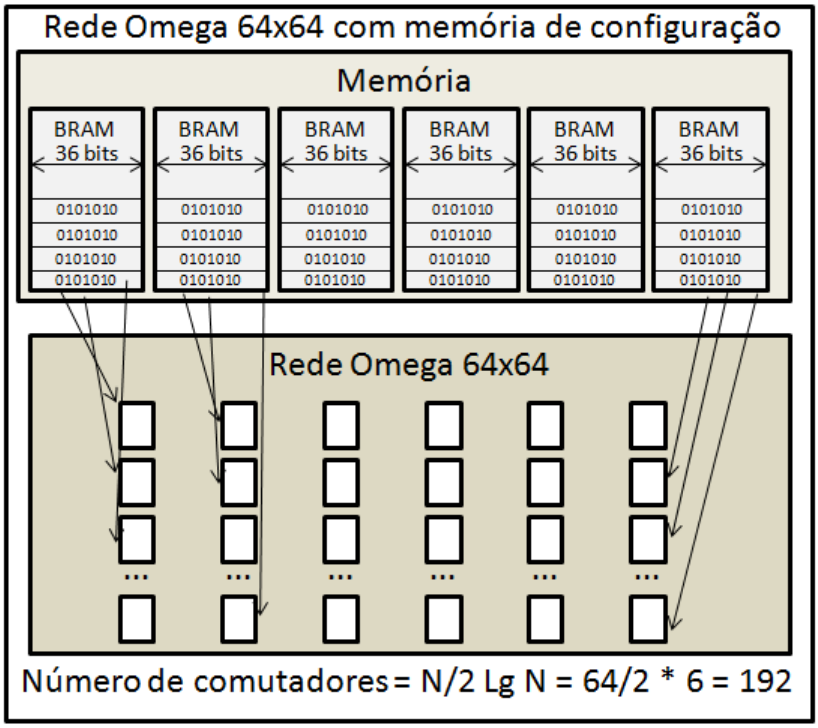


Figura 3.6: Rede Omega de tamanho 64x64 utilizando 6 módulos de memória(BRAM)

	Número de entradas				
	32	**64**	**128**	**256**	**512**
Nº de Módulos	0	6	14	32	72

Tabela 3.3: Módulos de Memória sintetizado em uma Virtex6 com 416 módulos.

3.2.4 Rede Multiestágio e Unidades Funcionais

Diferente dos experimentos anteriores, onde a rede foi sintetizada isoladamente, no terceiro experimento, a rede é usada para interconectar um conjunto de unidades funcionais ou ALU como ilustrado na Figura 3.7. O objetivo é avaliar a capacidade da ferramenta de síntese na geração de um circuito com unidades funcionais interligadas através da rede multiestágio e comparar os resultados da estimativa da síntese da rede em separado. Cada ALU implementa as seguintes operações: soma, subtração, multiplicação, deslocamento e funções lógicas como E, OU e negação. A Tabela 3.4 apresenta resultados de síntese com ALUs com a largura de bits da palavra de 8, 16 e 32 bits conectadas a uma rede com 32 entradas. Como cada ALU tem duas entradas e uma saída, a rede tem as 32 saídas conectadas as entradas das ALU, e as 16 saídas das ALUs juntamente com 16 entradas externas são conectadas a entrada da rede (Figura 3.7). As entradas externas servem para alimentar as ALUs. Podemos observar na Tabela 3.4, que o tamanho da rede em separado pode ser usado com uma base de estimativa da área ocupada. A Figura 3.7(a) ilustra a síntese em separado dos elementos. Ao realizar a síntese em conjunto, a área total foi próxima da estimativa da soma em separado da área das ALUs e da rede, como podemos ver na Figura 3.7(b).

Largura	**8bits**	**16 bits**	**32 bits**
Área em LUTs das 16 ALUs	1824	3824	7920
Área em LUTs 16 ALUs + Rede sintetizado em separado	2848	5872	12016
Área em LUTs das ALUs + Rede sintetizado em conjunto	2910	6187	12228

Tabela 3.4: Área das ALU sintetizadas em separado e em conjunto com uma rede Omega de 32 entradas.

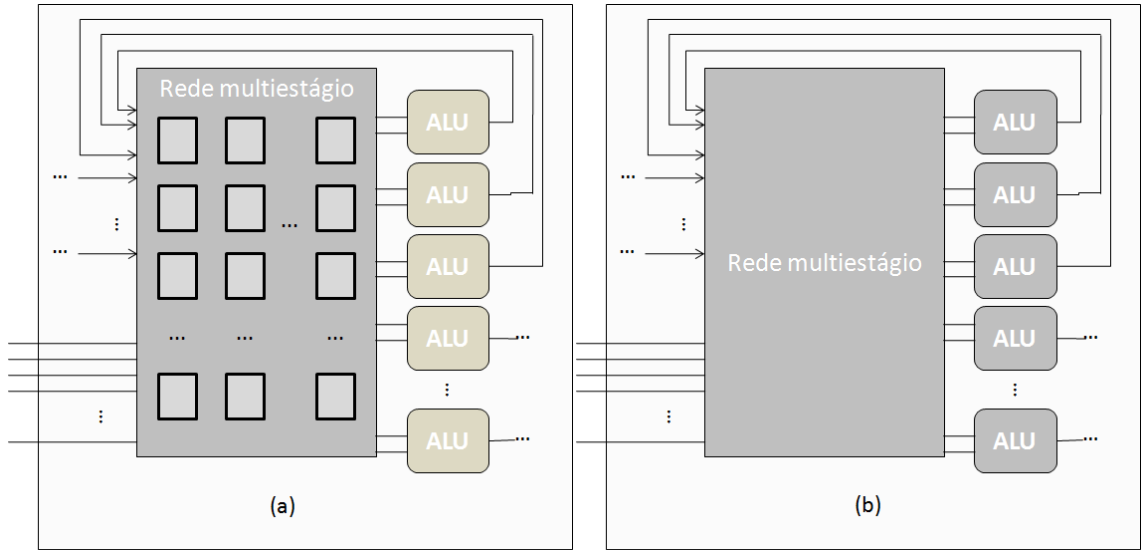


Figura 3.7: Rede multiestágio interconectando ALUs; (a) síntese em separado; (b) síntese de todo o conjunto.

3.2.5 Registros entre Estágios

No quarto experimento foram introduzidos registros entre os estágios da rede para avaliar o atraso e tamanho em aplicações onde se pode usar a rede em modo *pipeline*. A Tabela 3.5 mostra os resultados de área e latência para a rede com registros entre os estágios. A vantagem é a redução do tempo de relógio para 0,5 ns. O aumento médio em área foi de 30% para uma rede com largura da palavra de 8 bits. Os módulos de memória podem armazenar a configuração da execução em *pipeline*, aumentando a vazão dos resultados como ilustrado na Figura 3.8.

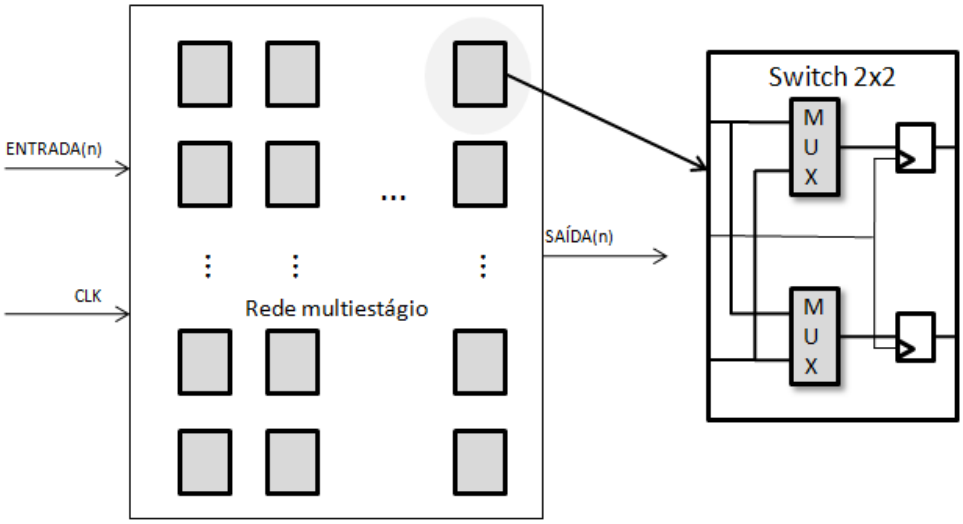


Figura 3.8: Rede multiestágio com registros nas colunas

Tamanho Rede Omega com Pipeline (LUTs)					
	Número de entradas				
	N = 32	**N = 64**	**N = 128**	**N = 256**	**N = 512**
Área com *pipeline*	1549	3370	7403	17104	37294
Área sem *pipeline*	1227	2443	6039	12592	29107
Acréscimo	26,00%	38,00%	23,00%	36,00%	28,00%
Atraso em ns	0,52	0,52	0,52	0,52	0,52

Tabela 3.5: Rede com *Pipeline* (Registros entre os estágios). Largura de 8 bits

3.2.6 Radix do Comutador

No quinto experimento, o tamanho do comutador foi avaliado. Ao aumentar o radix do comutador, o número total de comutadores é reduzido, porém o tamanho do comutador aumenta. O custo total da rede irá depender da tecnologia onde a rede será implementada. Neste experimento, a tecnologia alvo é uma Virtex6 com LUT de 6 entradas. O tamanho da radix2 evolui com O(N Lg N) como comprovado no experimento 1, mostrando que a síntese em FPGA não aumenta a complexidade de custo em área da rede. Já o padrão radix4 com comutadores 4x4 reduz a área total ocupada com a redução do número de comutadores para N/4 Lg_4N. O aumento da complexidade do comutador não influenciou a implementação. Podemos ver a diferença entre uma rede Omega 16x16 radix2 e radix4 na Figura 3.9

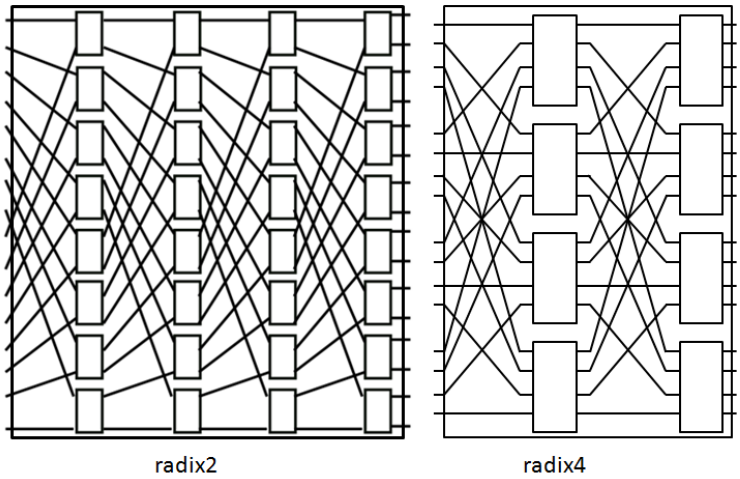


Figura 3.9: Rede Omega 16x16 utilizando radix2 e radix4

Tamanho e Latência das Redes radix2 e radix4 em LUTs				
	Número de entradas			
	32	**64**	**128**	**256**
1 bit, Radix2 – Área (LUTs)	392	425	1108	1838
1 bit, Radix4 – Área (LUTs)	NA	192	NA	1024
32 bits, Radix2 – Área (LUTs)	4299	9355	22765	49456
32 bits, Radix4 – Área (LUTs)	NA	6144	NA	32768
Radix2 – Latência (ns)	1,97	2,27	2,64	3,20
Radix4 – Latência (ns)	NA	2,08	NA	2,69

Tabela 3.6: Tamanho Rede Radix 4 e Radix 2. Área em LUTs e atraso em ns.

A Tabela 3.6 apresenta os resultados para síntese com radix4 para uma rede de 1 bit de largura e de 32 bits de largura. Os resultados são comparados com a rede radix2. Em alguns casos não se aplica (NA), pois o radix4 tem que ser potência de 4 como já mencionado. Para a rede com largura de 1 bit, a radix4 ocupa uma área em LUTs duas vezes menor. Para a largura de 32 bits, a área da radix4 é em torno de 60% da área da radix2. Esse ganho de tamanho da radix4 pode ser explicado pela arquitetura existente nas LUTs da Virtex6 onde cada uma possui 6 entradas, o que é o ideal para um mux 4-1 pois são quatro entradas de dados mais duas entradas de configuração, totalizando seis entradas, já a radix2 são duas entradas de dados e uma de configuração para cada mux2-1, assim as LUTs ocupadas seriam possivelmente subutilizadas. Ademais, a latência da rede é reduzida de 10 a 20% com o uso do radix4.

3.2.7 Rede Benes

No sexto experimento foi avaliada a síntese de outro padrão multiestágio, a rede Benes que usa o padrão *baseline* entre os estágios. Além disso, a rede Benes é rearranjável. A rede terá o dobro de estágios em comparação com uma rede Omega. Um código VHDL parametrizado também foi gerado. A Tabela 3.7 mostra os resultados para área em LUTs e o atraso em nanosegundos considerando uma largura de 8 bits para palavra. Podemos observar que ao dobrar o tamanho da rede, a latência aumenta em média 0,60 ns, pois em uma rede Benes, dobrar o tamanho significa incluir dois estágios a mais. Em relação à rede Omega bloqueante com largura de 8 bits ambas sem *pipeline* e memórias embarcadas, a área em LUTs é 70% maior. Para aplicações com compilação estática as redes Benes são interessantes, pois não geram conflitos e o roteamento tem complexidade O(N Lg N) e pode ser facilmente incorporado no compilador.

Rede Benes					
	Número de entradas				
	32	**64**	**128**	**256**	**512**
Área em Luts para 8 bits de largura	1792	4224	9984	23040	52224
Latência em ns	3.45	4,07	4,75	5,43	6,11

Tabela 3.7: Rede Benes. Largura de 8 bits. Área em LUTs e atraso em ns.

3.2.8 FPGA

No sétimo experimento, analisamos o comportamento do crescimento e tamanho das redes multiestágio em duas arquiteturas diferentes de FPGA, uma arquitetura utilizando LUTs de 4 entradas, utilizando um FPGA Spartan3 da Xilinx, e a arquitetura que já vem sendo usada nos testes anteriores, uma Virtex6 com LUTs de 6 entradas. Como podemos ver na Tabela 3.8 a rede radix4 aproveita-se das LUTs de 6 entradas na Virtex6. Vemos também que uma rede de 64 E/S é 6 vezes menor que uma rede *crossbar* e 50% menos que a rede radix2.

A Tabela 3.8 mostra o custo da área para redes de 16 bits de largura implementadas em dois FPGAs da Xilinx com arquiteturas distintas: Spartan3 e Virtex6. A primeira coluna mostra qual rede é. A segunda coluna mostra o tamanho do comutador em cada arquitetura. A última linha mostra o tamanho de uma rede *crossbar*, a rede *crossbar* foi implementada somente na Virtex6, podemos ver que esta rede pode ser mais de 10 vezes maior que uma radix4, sendo uma rede mais complexa, pois é uma rede não bloqueante podendo realizar qualquer ligação.

Tamanho das redes em LUTs								
	Spartan3				Virtex6			
Radix	Comutador	Número de entradas			Comutador	Número de entradas		
		16	64	256		16	64	256
2	32	2048	6144	32768	16	768	4540	24160
4	128	2048	6144	32768	64	512	3072	16384
Crossbar	-	-	-	-	-	1024	21344	278528

Tabela 3.8: Custo em área em LUTs para uma Spartan3 e uma Virtex6 com 16 bits de largura.

3.3 Considerações finais

Neste capítulo fizemos um estudo dos custos de implementação de redes multiestágios em FPGA. Vimos que as redes podem ser descritas com código parametrizável em VHDL. As ferramentas de síntese em FPGA capturam a regularidade da rede com a complexidade O(N Lg N) em área e em bits de configuração bem como a latência com complexidade O(Lg N). Os módulos de memória embarcados nos FPGAs de última geração constituem uma boa alternativa para armazenar internamente várias configurações (até 512 em uma Virtex6). Podemos melhorar também a frequência de operação utilizando pipeline entre os estágios das redes. O espaço ocupado pelas redes é reduzido e é viável a síntese de várias unidades de processamento simples como ALUs.

Vimos também que redes Omega usando radix4 são mais eficientes em questão de tamanho e possuem menores tempos de latência, além de se beneficiarem com a arquitetura da Virtex6 de LUTs de 6 entradas. Entretando, na radix4, o número de entrada das redes deve ser potência de 4.

4. Roteamento da Rede Omega

Neste capítulo serão apresentados dois algoritmos de roteamento em hardware para redes Omega e um estudo da capacidade de roteamento das redes. Estes resultados são a continuidade de trabalhos anteriores [Ferreira, 2009a][Ferreira, 2009b].

Um algoritmo em hardware para o roteamento em redes Omega foi apresentado em [Ferreira, 2009a]. O algoritmo propõe o uso de controle distribuído e um decodificador de prioridade. Entretanto, o algoritmo está preso ao contexto de uma arquitetura na qual apenas a entrada é escolhida. O algoritmo retorna a primeira saída que é roteável. Posteriormente em [Park, 2008], o algoritmo foi estendido para incorporar tolerância a falhas. Nesta dissertação, uma das contribuições é uma versão que executa o roteamento para um par entrada/saída com o decodificador de prioridade para redes com radix2. Ou seja, a saída também é especificada. Além disto, uma implementação baseada em memória é apresentada. Estes resultados foram publicados no trabalho [Vendramini, 2010b]. Além das duas versões, foi avaliado o custo de implementação dos algoritmos em FPGA. Os trabalhos anteriores tinham estimativas para implementações em VLSI [Ferreira, 2009a]. Outra contribuição dessa dissertação foi adaptar o algoritmo para radix4 e levar em consideração o *multicast* em uma arquitetura paralela com duas redes Omega e estágios extras, cujos resultados foram publicados em [Ferreira, 2011a].

Além dos algoritmos, este capítulo apresenta também um estudo da capacidade de roteamento das redes Omega em função da carga, do número de entradas, so número de estágios extras, do radix e do número de redes em paralelo.

Inicialmente, os trabalhos correlatos de roteamento são discutidos. Em seguida, o roteamento na rede Omega é detalhado. Depois, os dois algoritmos em hardware são apresentados. Posteriormente, a avaliação da capacidade da rede em função de diversos parâmetros é apresentada assim como os custos em atrasos e área da implementação em FPGA dos algoritmos de roteamento.

4.1 Trabalhos Correlatos

Nas arquiteturas paralelas com redes multiestágios que foram propostas nas décadas de 70/80, havia dois grandes gargalos. Primeiro, a configuração da rede com complexidade

O(N*Lg N). A grande vantagem da rede era seu atraso O(log N) e a sua estrutura paralela de comunicação. Porém para configurar a rede ou realizar sua inicialização (*setup*) eram necessários O(N*Lg N) passos. Vários autores propuseram algoritmos paralelos [Lee,1987] [Nassimi,1981] [Çam,1999]. O segundo gargalo era a construção física das máquinas paralelas. Ou seja, a maioria dos trabalhos tinham cunho teórico e raramente as arquiteturas eram fisicamente construídas. Atualmente, a maioria das arquiteturas construídas são paralelas. O que motiva a busca por mecanismos paralelos de comunicação como as redes multiestágio. Um exemplo recente é o uso da Benes em uma cache tolerante a falhas em nanotecnologia [Ansari, 2009]. Entretanto, o roteamento é feito estaticamente em software com algoritmos de coloração de grafos.

Vários algoritmos paralelos para redes Benes foram propostos [Lee,1987] [Nassimi,1981] [Çam,1999]. Um algoritmo com complexidade O(Lg n) foi proposto em [Nassimi,1981]]. Porém para configurar a rede o algoritmo tem que ser executado em uma arquitetura paralela onde todos os processadores estão ligados em todos os processadores, que tem custo $O(n^2)$ no mínimo. Ou seja, a análise de complexidade foi feita em alto nível, ao se implementar em uma máquina paralela real, outros pontos devem ser contabilizados. Não são apresentados tempos de execução nem consumo de recursos de memória, apenas a complexidade em um nível abstrato. Os outros trabalhos são semelhantes ao abordarem a complexidade em um alto nível [Lee,1987] [Çam,1999].

Recentemente em [Schäck, 2009], uma implementação com Omega em FPGA foi apresentada. A rede é usada para conectar um conjunto de processadores RISC através do acesso compartilhado de memória. Cada processador pode diretamente ler e gravar em sua memória local. Para acessar a memória de outro processador, a rede Omega é usada. Não se usa estágios extras, podem ocorrer vários conflitos. A implementação é feita em hardware usando redes radix2 com comutação de pacotes. A entrada superior do comutador tem prioridade no caso de conflitos. Como não se sabe se ocorrerão conflitos, a comunicação é sempre sincronizada no acesso paralelo as memórias com um comando em software. Como já foi mencionado, esta dissertação aborda a comunicação por circuito em níveis de granularidade menor que os processadores.

Outra questão fundamental é porque usar uma rede bloqueante. Quais os benefícios de redução de custo em relação a perda de conectividade. A capacidade de roteamento de uma rede bloqueante já foi muito estudada, mas ainda existem problemas em aberto, como a

modelagem analítica [Gazit, 1989]. Para uma rede com 32 entradas sem estágios extras, temos 10^{35} permutações completas onde apenas 10^{24} podem ser roteadas sem conflito. Se adicionar um estágio extra, teremos 10^{28} permutações roteadas segundo uma avaliação por amostragem apresentada em [Gazit, 1989]. Nossa abordagem, proposta aqui, avalia a capacidade com mais de um estágio extra, uma ou duas redes, radix2 e radix4. Além disso, verificamos qual a percentagem de ligações são roteadas com sucesso para as permutações parciais, onde apenas um subconjunto de entradas/saídas são roteadas.

4.2 O Roteamento

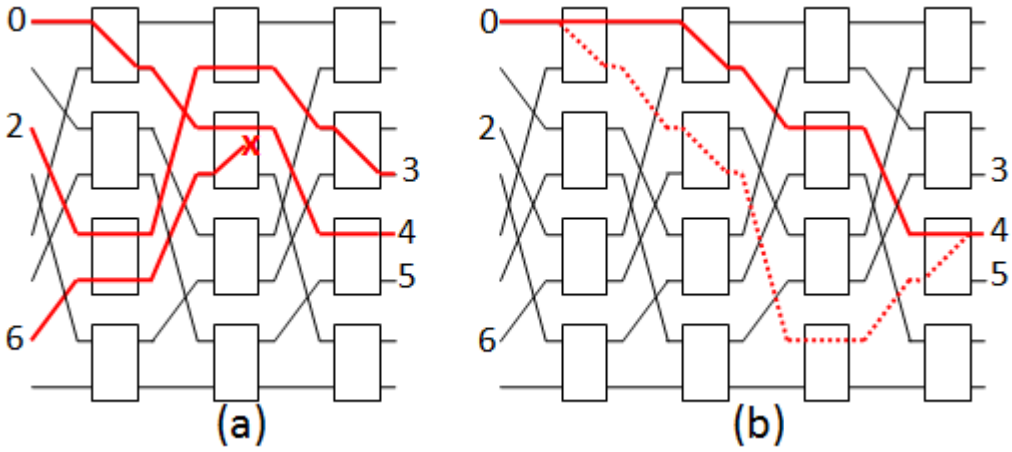


Figura 4.1: Redes: (a) Omega; (b) Omega com 1 estágio extra.

Como já falado anteriormente, a rede Omega sem estágios extras tem um único caminho para cada par de entrada e saída. O caminho pode ser determinado pelo XOR dos endereços de entrada e saída. O caminho passa por um comutador em cada estágio. Cada bit do XOR resultante irá programar um comutador. O bit mais (menos) significativo do XOR corresponde a configuração do primeiro (último) estágio. Para cada bit, se o resultado do XOR for 1 (0), o comutador do estágio correspondente é ligado cruzado (direto). A Figura 4.1(a) mostra um exemplo de roteamento da entrada 0 ou 000, na saída 4 ou 100 em binário, cujo XOR será 100, com os comutadores nos modos cruzado, direto e direto. A figura também ilustra a ligação da entrada 2 e a saída 3 cuja a programação dos comutadores é dada por 010 xor 011 = 001, ou seja, direto, direto e cruzado.

Um conflito ocorre quando dois pares de entradas/saídas passam pela mesma linha de um comutador em um determinado estágio. Para determinar se ocorrerão os conflitos tem-se que verificar cada nova ligação estágio por estágio. Uma solução é usar o roteamento com janelas. Onde a janela determina a linha em cada estágio. A palavra de roteamento *w* é formada pela concatenação dos endereços de entrada e saída. Uma janela de *LgN* bits irá se deslocar sobre a palavra de roteamento que tem 2 LgN bits. A primeira janela começa no segundo bit mais significativo. Para uma rede com 8 entradas, teremos $w = o_2o_1o_0d_2d_1d_0$. As janelas serão = $o_2\underline{o_1o_0}d_2d_1d_0$, = $o_2o_1\underline{o_0d_2}d_1d_0$, = $o_2o_1o_0\underline{d_2d_1d_0}$. O par entrada/saída 0 -> 4 *w* = 000100 e as janelas *001, 010* e *100*. Neste exemplo, no primeiro estágio passará pela linha 1, depois pela linha 2 e finalmente a linha 4 de destino (veja Figura 4.1(a)). Quando outro par entrada e saída tenta se conectar pode ocorrer um conflito se for passar pela mesma linha no mesmo estágio. Por exemplo, o par 6 -> 5 que gera a palavra *w* = 110101, irá conflitar no segundo estágio na linha 2 onde *w* = 11<u>01</u>01.

Uma maneira de reduzir conflitos é a adição de estágios extras. Cada estágio irá dobrar o número de caminhos. A Figura 4.1(b) ilustra uma rede Omega com um estágio extra. A ligação *0 -> 4* terá dois caminhos. A palavra de roteamento para *n = 8* será $w = o_2o_1o_0x_0d_2d_1d_0$, concatenando a entrada, o estágio extra e a saída. O bit extra x_0 pode ser 0 ou 1, gerando as duas opções. O algoritmo é o mesmo, levando um estágio a mais em consideração. No exemplo da ligação 0 -> 4, teremos a opção com $x_0 = 0$, com *w = 000<u>0</u>100* e as janelas (ou linhas) *000, 001, 010 e 100*. Para a segunda opção temos $x_0 = 1$, com *w = 000<u>1</u>100* e as janelas (ou linhas) *001, 011, 110 e 100,* como mostra a Figura 4.1(b) em pontilhado. Com o estágio extra, podemos rotear também a ligação *6 -> 5* sem conflito com a ligação *0 -> 4*. Suponha o bit extra $x_0 = 0$ para a ligação *0 -> 4,* desenhada na Figura 4.1(b). Para ligar *6 -> 5* teremos que avaliar as duas opções. Para $x_0 = 0$ teremos com *w = 110<u>0</u>101* e as janelas *100, 001, 010 e 100*. O conflito continua, agora na linha 1 do segundo estágio. Já para a segunda opção com $x_0 = 1$, teremos com *w = 110<u>1</u>101* e as janelas *101, 011, 110 e 101,* não ocorrerão conflitos. A implementação do algoritmo em software é direta, para realizar as ligações sob demanda. Todas as possibilidades para os estágios extras são verificadas sequencialmente até ter sucesso. A complexidade para *k* estágios extras será *O(Lg n)* para gerar as janelas de cada tentativa e $O(2^k)$ tentativas no pior caso, totalizando $O(2^k \text{ Lg } n)$. O pseudo código do algoritmo é ilustrado na Figura 4.2.

```
Parâmetros:
N: tamanho da entrada e da saída, M: número de estágios = Log2(N) + K
K: número de estágios extras, L: 2^k
Bitmask: comprimento Log2(N), isto é: se N=8 então bitmask = 111

RoteamentoOmega(entrada, saida)
1. F1: for  extra <= 0 to L do //testa todas as possibilidades para estágios extras
2.    caminho <= (entrada << M) OR (extra << log2 (N)) or saida
3.    encontrou <= true
4.    for j <= 0 to log2(N)+k-1 do // do estágio 0 até o estágio Log2(N)+k-1
5.       i <= caminho >> (Log2(N)+k+1-j) AND bitmask
6.       encontrou <= encontrou AND linhaLivre[i][j] // verifica se a conexão na linha é livre
7.       if encontrou then
8.          linha[j] <= i
9.       else   continue F1
10.         end if
11.   end for
12.   if encontrou then
13.     linhaLivre[linha[0:Log2(N)+k-1]][j] <= false // coloca a linha como ocupado
14.     return true
15.   end if
16. end for
17. return false //conflito no caminho
```

Figura 4.2: Algoritmo de roteamento da rede Omega radix2 com estágios extra.

Vale ressaltar que os conflitos dependem da ordem que os pares são roteados e do código atribuído. Os algoritmos para redes rearranjáveis verificam simultaneamente todos os pares. Neste trabalho, abordamos as redes Omega com estágios extras com um algoritmo de roteamento guloso que é executado sob demanda para aplicações onde não conhecemos todas as ligações a priori. Ou seja, a rede começa sem nenhuma ligação e a cada passo uma nova ligação é adicionada.

A seguir, os dois algoritmos propostos nesta dissertação serão apresentados. O primeiro é uma versão paralela em hardware do pseudo código da Figura 4.2, onde as linhas 4 à 11 são verificadas em paralelo e também as linhas 12 a 14 são feitas em paralelo. A segunda versão avalia todas as possibilidades em paralelo com uma implementação de controle distribuído.

4.3 O algoritmo usando memórias embarcadas

Uma versão paralela do algoritmo de roteamento com janela de deslocamento (Figura 4.2) é proposta e implementada em hardware com o uso de módulos de memória por estágio. A Figura 4.3 apresenta a estrutura geral da implementação. A palavra de roteamento é construída concatenando os endereçamentos de entrada, estágios extras e saída. O código dos estágios extras vem de um registrador que é inicializado com zero. Se não for possível rotear com o código zero, o registrador é incrementado e o novo roteamento é verificado, a cada ciclo de relógio. Os bits de configuração dos comutadores são armazenados em memória separadas, permitindo que em apenas um passo, em paralelo, todos os pares linha/coluna gerados pelas janelas de roteamento sejam verificados. No melhor caso, o algoritmo executa em um passo. Para o pior caso, todos os códigos são verificados, portanto para *k* estágios extras, serão executados no máximo 2^k passos.

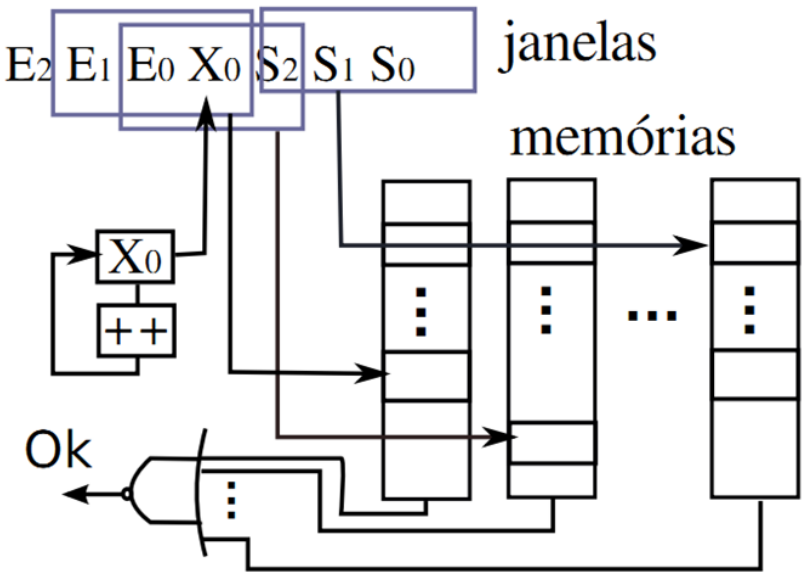


Figura 4.3: Arquitetura do algoritmo utilizando janela em memórias

Cada módulo de memória deve armazenar dois bits. Um bit para dizer se a linha do comutador está ocupada ou livre. Caso esteja ocupada, o segundo bit irá dizer a direção da configuração se é direta ou cruzada. Para *unicast* se usa apenas um bit para dizer a direção, pois só existem duas possibilidades, 0, para ligações diretas, isto é, 0 -> 0 e 1 -> 1, e 1, onde teremos ligações cruzadas, 0 -> 1 e 1 -> 0.

Como exemplo, utilizaremos uma arquitetura onde a rede Omega possui tamanho 8x8, mais um estágio extra, com a ligação 2 -> 3, já realizada. Iremos realizar a conexão 0 -> 4, pode-se ver na Figura 4.4(a) que a memória já está preenchida com a conexão 2 -> 3, os espaços em branco estão livres, então o valor preenchido é 0 0. Lembrando que o primeiro bit informa se o caminha está livre se for 0 e ocupado se for 1, e o segundo bit informa se a ligação é cruzada(1) ou direta(0). A Figura 4.4(b) mostra o funcionamento do algoritmo de roteamento por janelas, cada coluna acessa uma linha em paralelo e verifica se a linha está livre ou ocupada, se alguma estiver ocupada, o roteamento falha. Caso contrário, o roteamento da ligação é realizada.

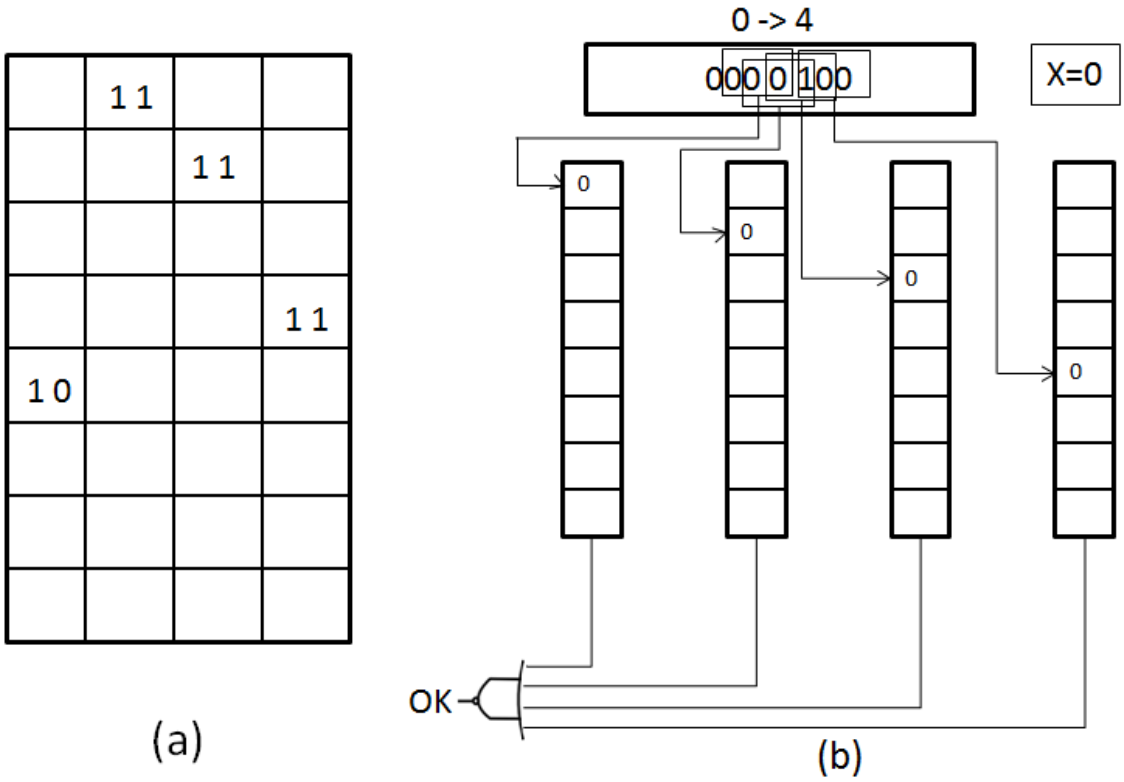


Figura 4.4: (a) Memória de configuração da arquitetura de roteamento – radix2; (b) Funcionamento da arquitetura por janelas usando memórias fazendo o roteamento de uma conexão – radix2.

Porém a versão implementada foi a utilizando *multicast* onde são necessários dois bit de configuração, um bit para cada saída do mux, mais 1 bit de ocupado. No algoritmo de *multicast* como uma entrada pode se propagar em várias saídas livres, o algoritmo verifica se alguma parte do caminho está ocupada, se todo o caminho estiver livre, ele cria a conexão, mas se alguma parte estiver ocupada, ele verifica se esta parte ocupada está ocupada com a entrada que ele deseja usar, para isso o algoritmo verifica o bit de configuração da saída da coluna

anterior. Se a configuração já gravada for igual à configuração da conexão que o algoritmo está querendo rotear, ele percebe que o caminho é o mesmo e aceita a conexão, caso contrário a ligação não pode ser completada.

Para o radix4 utilizando *multicast*, a ideia é semelhante ao radix2, porém a diferença está no número de bits de configurações, cada saída do mux, possui 2 bits, temos 4 saídas, então teremos um total de 8 bits. Os bits informam em qual entrada a saída está conectada. Para realizar o roteamento com *multicast*, em cada coluna é verificado se o caminho está livre, se está, a conexão é feita, caso não esteja, é verificado se a coluna ocupada pode ser usada, caso a conexão seja da mesma entrada, para isso verifica-se a saída da coluna anterior, e se os bits de configuração já gravados forem iguais aos bits de configuração que o algoritmo esta querendo rotear (os dois bits a esquerda da janela verificada), a ligação é completada. Caso contrário, a ligação não será feita.

Como exemplo, a Figura 4.5(a) ilustra uma memória de uma rede radix4 16x16 com a ligação 5-> 2 já realizada. A Figura 4.5(b) ilustra o algoritmo de roteamento tentando rotear a ligação 10 -> 3. Porém a mesma gerará um conflito com o estágio extra 0. Tomando 10(1010) e 3(0011) e extra 0 (00), juntando os bits teremos 1010000011. Assim, teremos as seguintes janelas: 10<u>10000</u>011, 1010<u>000011</u>, 1010<u>000011</u>. A janela do meio dará conflito com a ligação 5 -> 2, como podemos ver na Figura 4.5(a), verificamos então se os bits de configuração do mux são iguais, para isso verifica-se os dois bits a esquerda da janela com conflito. O valor obtido é <u>10.</u> Que é diferente do que está na memória(<u>01</u>). Assim o algoritmo acusará um conflito, e incrementará o estágio extra para 1(01). A Figura 4.6(a), exibe o algoritmo de roteamento, com o estágio extra incrementado em 1. E dessa vez não havendo conflito. A Figura 4.6(b) exibe a memória de configuração com a ligação 10 -> 3, já realizada. Tomando 10(1010) e 3(0011) e extra 1 (01), juntando os bits teremos 1010010011. Assim, teremos as seguintes janelas: 10<u>10010</u>011, 1010<u>010011</u>, 1010010<u>0011</u>. Assim, os bits de configuração serão: 10 10 01.

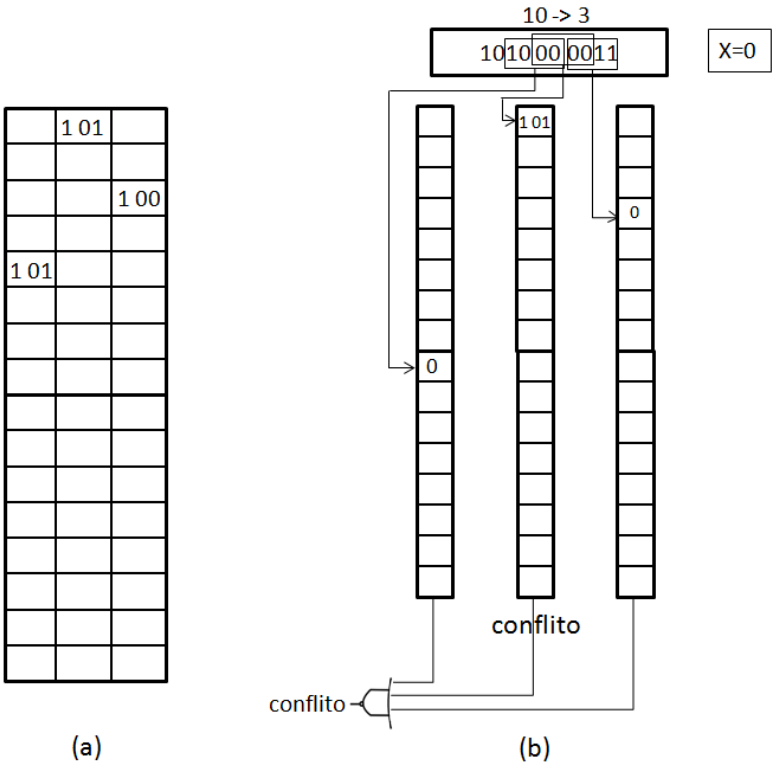


Figura 4.5: (a) Memória de configuração da arquitetura de roteamento – radix4; (b) Funcionamento da arquitetura por janelas usando memórias gerando conflito no rotemento – radix4.

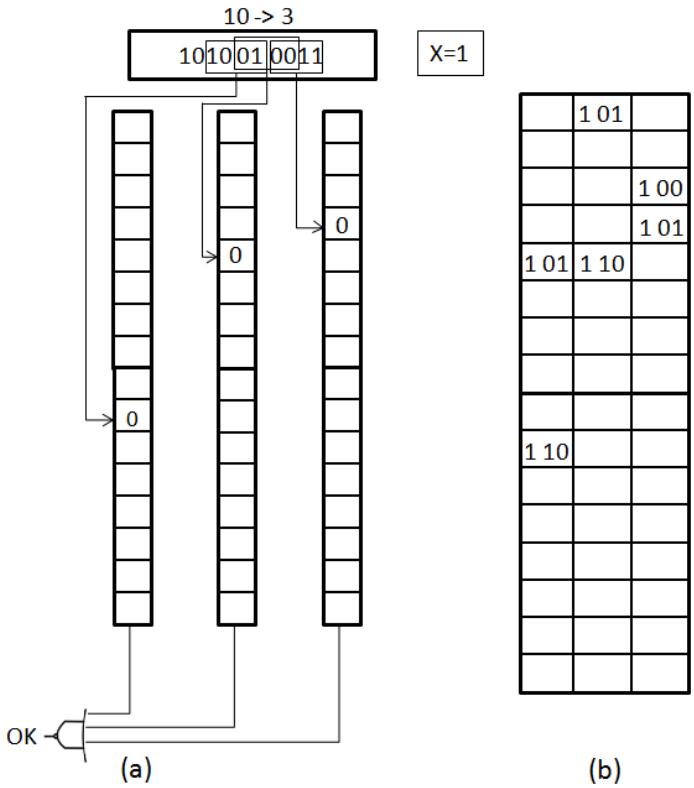


Figura 4.6: (a) Arquitetura roteando a ligação 10 -> 3 com extra = 1; (b) Memória com as ligações 5 -> 2 e 10 -> 3.

As redes Omega possuem a vantagem de ter *Lg n* estágios. Entretanto estas redes são bloqueantes. Uma rede rearranjável tem o dobro de estágios como mencionado no Capítulo 3. Outra alternativa, equivalente em custo a uma rede BENES com *2 Lg n,* são duas redes Omega em paralelo como ilustra a Figura 4.7. Apesar de continuar sendo uma rede bloqueante, o atraso será reduzido pela metade em comparação com a rede rearranjável.

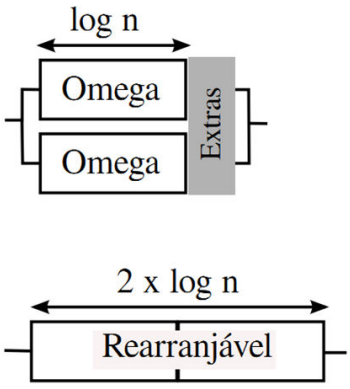


Figura 4.7: Duas Redes Omegas em Paralelo VS. Rede Rearranjável.

Além disso, o algoritmo de roteamento em memória pode verificar o roteamento nas duas redes em paralelo. A implementação é simples, o algoritmo verifica se é possível rotear o caminho nas duas redes ao mesmo tempo, feito isso ele dá prioridade para a rede 0, caso ambas possam rotear o caminho, ele roteia o caminho na rede 0. Se somente a rede 1 puder rotear o caminho, ele grava o caminho na rede 1. Como ilustra o pseudocódigo da Figura 4.8

```
1   Roteamento(entrada, saida)
2       Para extra de 0 até NEstExtras
3       Faça:
4           criaJanelas(entrada, extra, saida);
5           Em paralelo:
6               verificaRoteamentoRede0();
7               verificaRoteamentoRede1();
8           Se !conflitoRede0
9               roteia na rede0;
10              retorna roteou com sucesso;
11          Senão Se !conflitoRede1
12              roteia na rede1;
13              retorna roteou com sucesso;
14          Senão continua for;
15      Fim for;
16      Retorna erro de rotemento;
17  Fim Roteamento;
18
```

Figura 4.8: Pseudocódigo algoritmo de roteamento das redes em paralelo.

Em trabalhos anteriores [Ferreira, 2009b][Ferreira, 2011b], duas redes Omega em paralelo eram usadas. Porém, primeiro era verificado a rede 0, em caso de insucesso, o estágio era incrementado. Como ilustra o Pseudocódigo da Figura 4.9 do Isto, gerava uma sobrecarga na rede 0. A abordagem proposta aqui verifica a rede 0 e a rede 1 para o mesmo código de estágio extra. Além disso, como já mencionado é uma versão em hardware que foi paralelizada e implementada em FPGA.

```
1   Roteamento(entrada, saida)
2       Para extra de 0 até NEstExtras
3       Faça:
4           criaJanelas(entrada, extra, saida);
5           verificaRoteamentoRede0();
6           Se !conflitoRede0
7               roteia na rede0;
8               retorna roteou com sucesso;
9       Senão continua for;
10
11      Para extra de 0 até NEstExtras
12      Faça:
13          criaJanelas(entrada, extra, saida);
14          verificaRoteamentoRede1();
15          Senão Se !conflitoRede1
16              roteia na rede1;
17              retorna roteou com sucesso;
18          Senão continua for;
19      Fim for;
20      Retorna erro de rotemento;
21  Fim Roteamento;
22
```

Figura 4.9: Pseudocódigo algoritmo de roteamento das redes sobrecarregando a rede 0.

4.4 O algoritmo usando codificador de prioridade

Um algoritmo de roteamento em hardware para redes Omega com estágios extras foi proposto em [Ferreira, 2009a] onde para uma dada entrada, o roteamento com *multicast* busca quais as saídas estão disponíveis. O algoritmo verifica em apenas um ciclo se existe um roteamento mesmo em presença de estágios extras, ao usar codificadores de prioridade. A latência do ciclo de roteamento depende em grande parte do codificador de prioridade. Como a rede está embarcada em uma arquitetura com várias saídas equivalentes, o algoritmo buscava a primeira saída livre.

A implementação que iremos apresentar difere em três aspectos. Primeiro, o algoritmo procura o roteamento para um par entrada/saída. Segundo, o algoritmo considera ligações *unicast*, ou seja, para uma entrada, só poderá haver uma saída. O que simplifica a estrutura interna, porém pode ser estendido para o caso de *multicast*. Terceiro, como a saída da rede é fixa, só é necessário usar um codificador de prioridade, a implementação proposta em [Ferreira,2009a] utiliza dois codificadores. Finalmente, um codificador de prioridade parcial é utilizado, reduzindo significativamente o atraso e custo. Além de uma implementação em FPGA ser apresentada, enquanto que a implementação anterior foi avaliada somente em VLSI.

A Figura 4.10 mostra a estrutura geral do roteamento composta por uma rede de unidades de controle distribuídas e interligadas seguindo o padrão da rede Omega. Cada comutador tem uma unidade de controle associada. A unidade armazena localmente o estado do comutador (ocupado ou livre). Dada uma entrada e_i, um decodificador irá injetar um sinal ativo (nível lógico 1) na linha *i*. As outras entradas irão receber o sinal inativo ou zero. Este sinal irá propagar em *broadcast* por *Lg n* estágios, da esquerda para direita. Ao mesmo tempo, na saída s_j, um segundo decodificador irá propagar um sinal ativo a partir da linha *j* no sentido oposto nos estágios extras. Os sinais de entrada e saída após propagarem pelas unidades de controle, cuja estrutura interna é ilustrada na Figura 4.11, são validados por linha em um codificador de prioridade no centro da estrutura. Se a linha de entrada do codificador estiver ativa, significa que existe um caminho livre. Pode haver mais de um, por isso é usado um codificador de prioridade. A saída do codificador indica os bits para programação dos estágios extras. Diferente do algoritmo com módulos de memória, onde para *k* estágios extras, são necessárias até 2^k tentativas, em apenas um ciclo, todas as possibilidades de roteamento são verificadas em paralelo e apenas uma é selecionada pelo codificador.

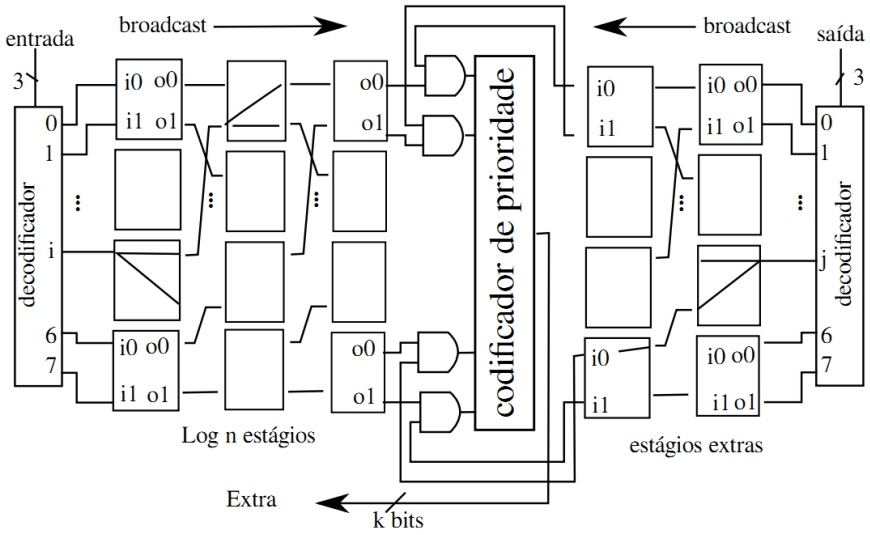


Figura 4.10: *Broadcast*

A Figura 4.11 mostra a estrutura interna das unidades de controle. A Figura 4.11 (a) mostra a unidade para propagar o *broadcast* nos *Lg n* estágios. Cada unidade é responsável por um comutador. O bit $busy_i$ é ativo quando a saída *i* do comutador está em uso. Se um sinal ativo chega à entrada da unidade, o sinal será propagado para as saídas de controle livres. A Figura 4.11(b) ilustra a unidade dos estágios extras que recebe um sinal de controle da saída. A ideia é semelhante, só que no sentido inverso, a saída recebe um sinal e se estiver livre será propagado para as entradas.

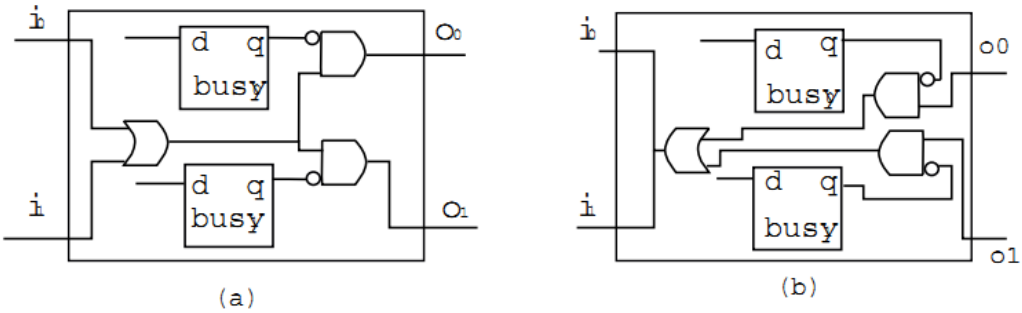


Figura 4.11: Unidades de Controle: (a) Lg n estágios; (b) extras.

Para um codificador de prioridade, dado *n* linhas de entradas de *1* bit cada, a saída indicará o código binário da primeira entrada ativa. Por exemplo, suponha um codificador com *16* entradas. Suponha que as linhas *3, 5* e *12* estão ativas. A saída indicará *0011*, ou seja, a linha *3*.

A Figura 4.12(a) apresenta o codificador de prioridade parcial. No caso de um decodificador parcial, a saída apresenta apenas os *m* bits mais significativos do código. No nosso exemplo, suponha que sejam *2* bits, a saída indicará *00* para a linha *3*. Seja *p = n - m* o número de bits menos significativos que são descartados. Um primeiro plano de *OR* agrupa as linhas de *p* em *p*. As saídas do plano de *OR* são conectadas a um codificador de prioridade completo de *m* bits que determina o primeiro grupo de linhas que está ativo nas entradas. Um codificador completo terá uma grande área e latência se sintetizado a partir de uma descrição em alto nível em VHDL ou Verilog nas atuais ferramentas de síntese (Xilinx/Altera). Recentemente em [Mohan, 2006], uma arquitetura com baixa latência para codificadores foi proposta para implementação em VLSI, usando *pass-transistors* e técnicas de propagação de *lookahead*.

No roteamento da rede Omega é possível utilizar o codificador parcial para encontrar os caminhos livres, pois o número de caminhos existe em relação ao número de estágios extras, então para um estágio extra, basta ter um codificador de prioridade que tenha apenas 1 bit, já que saberemos através dele se foi o caminho 0 ou o caminho 1 se está livre. Para 2 estágios extras, teremos 4 caminhos, então um codificador parcial de 2 bits que mostre qual caminho está livre é suficiente, e assim sucessivamente para o número de estágios extras.

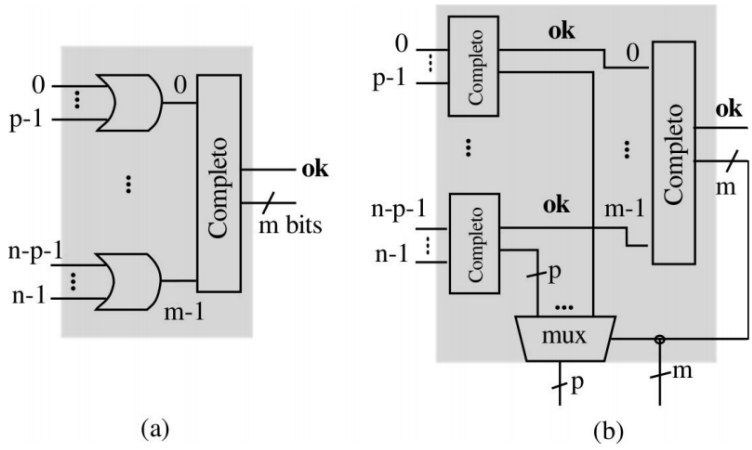


Figura 4.12: Codificador Prioridade: (a) Parcial (b) 2 Níveis.

Neste trabalho, diferente dos anteriores, abordamos a síntese em FPGA, onde outras considerações devem ser feitas. Uma técnica simples para otimizar o codificador e manter uma descrição parametrizada em VHDL é a decomposição em níveis, como ilustrada na Figura

4.12(b) para 2 níveis. Seja *n* linhas de entrada, onde *n = m + p*. O primeiro nível agrupa as entradas em *m* codificadores completos de *p* bits cada. O segundo nível recebe um bit de cada destes codificadores em um codificador completo de *m* bits que irá selecionar o primeiro grupo ativo do nível anterior. A saída do segundo codificador gera os *m* bits mais significativos e aciona um multiplexador que seleciona os *p* bits menos significativos do primeiro nível. Codificadores de prioridade são usados em memórias associativas. Uma implementação de memória associativa para FPGA é apresentada em [McLaughlin, 2009], porém o codificador de prioridade não é implementado devido a problemas de atraso. Neste trabalho mostramos que uma simples decomposição em níveis reduz significativamente o atraso e viabiliza a implementação.

4.5 Resultados experimentais

Trabalhos anteriores mostraram que redes multiestágio são alternativas interessantes para sistemas embarcados paralelos em FPGA [Vendramini, 2010a] [Neji, 2008]. Por exemplo, uma rede para interligar *64* unidades de *32* bits ocupa apenas *6%* e *4%* da área de um FPGA Virtex6 usando redes radix2 e radix4, respectivamente, como vimos no Capítulo 3. A latência da rede é de apenas *2* à *3 ns*. Seu custo é proporcional a *O(n Lg n)* e atraso *O(Lg n)*, e sua regularidade é capturada pelas ferramentas de síntese de FPGA.

Entretanto se faz necessário a implementação eficiente de algoritmos de roteamento para utilizar os benefícios da área e atraso das redes multiestágio. Nos resultados que serão apresentados, abordaremos: o tempo de execução e custo em hardware de duas implementações de roteamento. Será também apresentado um estudo da capacidade de roteamento das redes em função da carga das mesmas.

4.5.1 Capacidade de Roteamento Radix2

No primeiro experimento será avaliada a capacidade de roteamento parcial e completa com redes Omega com radix2. Por exemplo a Figura 4.13(a) mostra o teste com N=64, carga de trabalho de 50% com uma rede radix2 e dois estágios extras. A Figura 4.13(b) mostra um teste com duas redes radix4 com N=64, e um estágio extra em cada rede. Cada estágio extra em uma rede radix4 equivale a dois estágios extras em redes radix2. Isto acontece porque

adicionando um extra em radix2 dobra-se o número de possibilidades de caminhos, já com radix4 adicionando um extra, quadruplica-se o número de caminhos. Da Figura 4.14 à Figura 4.22 apresenta-se uma avaliação da capacidade de roteamento de uma e de duas redes Omega com e sem estágios extras. Diferente de trabalhos anteriores, os resultados apresentados aqui fazem uma avaliação em função da carga de trabalho, que é a quantidade de conexões que serão realizadas na rede.

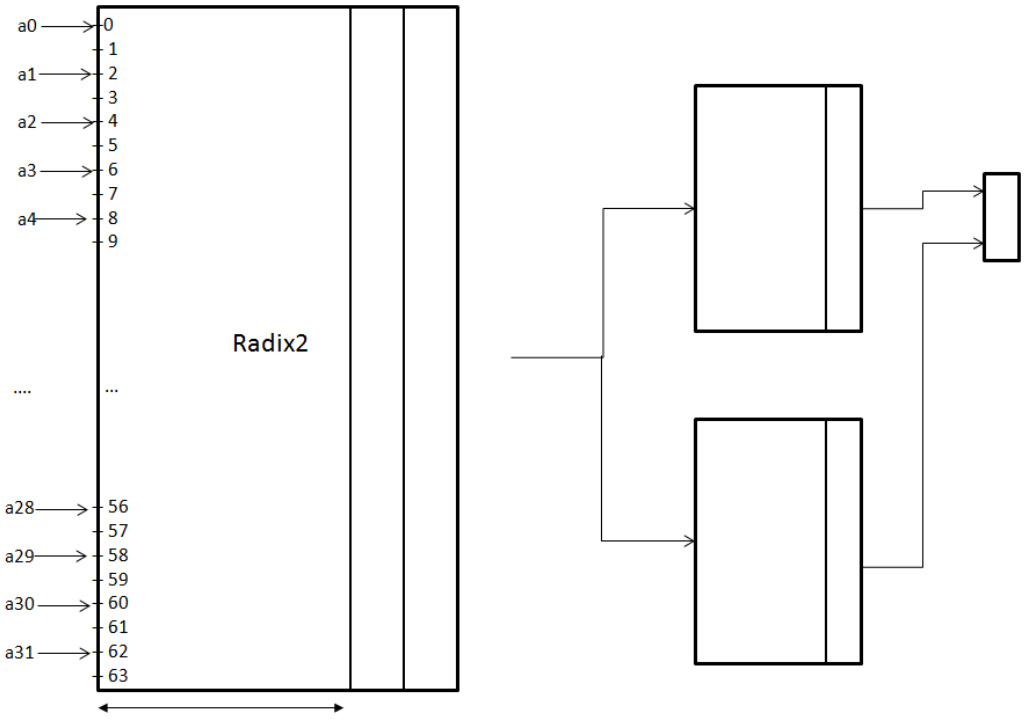
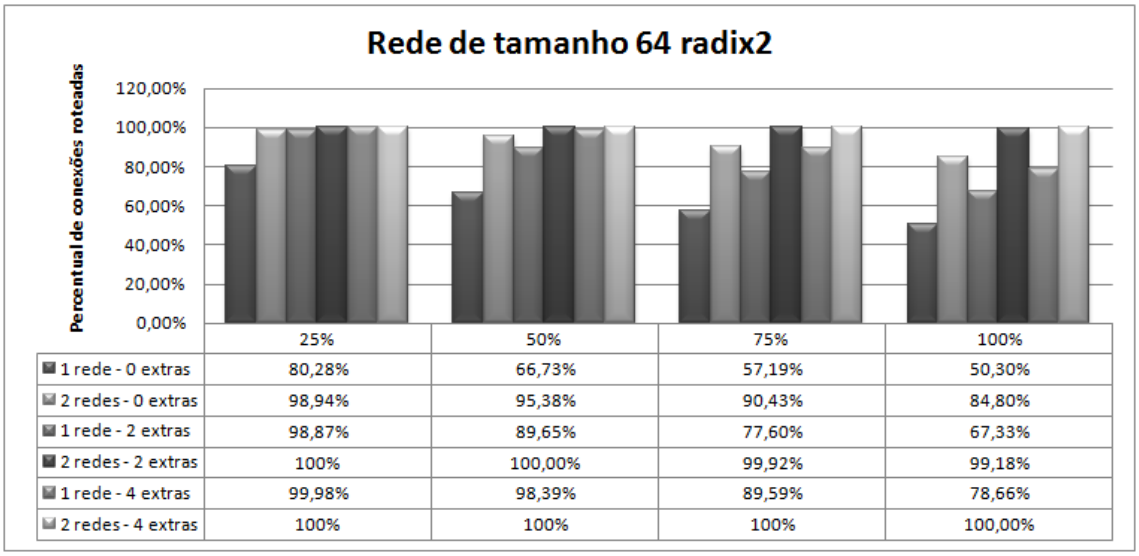


Figura 4.13: (a) Rede Radix2 com 2 extras, com 50% de carga de trabalho; Duas redes de tamanho 64 radix4 em paralelo com um estágio extra.

Quatro cargas de trabalho são mostradas nos gráficos, três com permutações parciais com 25%, 50% e 75% das ligações e uma configuração com permutação completa. A capacidade de roteamento foi avaliada por amostragem, foram geradas 100.000 permutações aleatórias para cada carga de trabalho, por exemplo, para uma rede de 32 entradas com 25% de carga, eram sorteadas 8 ligações possíveis aleatoriamente para representar uma permutação. O gráfico em barras serve para uma comparação visual dos casos estudados. Os valores na tabela abaixo de cada gráfico permite observar com mais precisão, os resultados do roteamento.

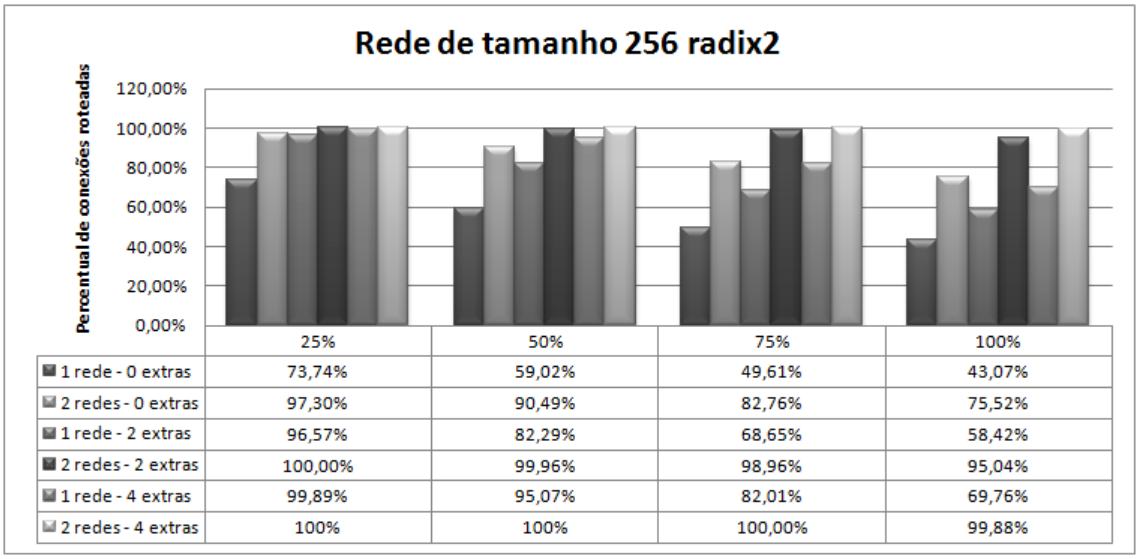
Quatro tamanhos de redes foram avaliados: 64, 256, 512 e 1024. A Figura 4.14, contém redes de tamanho 64. Como podemos ver o percentual de conexões realizadas diminui com a carga de trabalho da rede, na rede de 64 o pior caso encontrado foi 50,30% das permutações de uma carga de trabalho em 100% sem estágios extras, isso quer dizer que apenas metade das conexões puderam ser roteadas. Já com duas redes e 4 estágios extras, todas as conexões foram roteadas com sucesso para todas as cargas testadas. Podemos observar também na Figura 4.14, que utilizando uma rede com 4 caminhos extras e carga de trabalho de 25%, tivemos o valor de 99,98% das conexões roteadas com sucesso, isto quer dizer que das 100.000(permutações aleatórias)*64(tamanho rede)*0,25(carga de trabalho) = 1.600.000 tentativas de conexões, 1599680 conexões puderam ser roteadas com sucesso.



	25%	50%	75%	100%
1 rede - 0 extras	80,28%	66,73%	57,19%	50,30%
2 redes - 0 extras	98,94%	95,38%	90,43%	84,80%
1 rede - 2 extras	98,87%	89,65%	77,60%	67,33%
2 redes - 2 extras	100%	100,00%	99,92%	99,18%
1 rede - 4 extras	99,98%	98,39%	89,59%	78,66%
2 redes - 4 extras	100%	100%	100%	100,00%

Figura 4.14: Roteamento rede de tamanho 64, com uma e duas redes, com e sem estágios extras.

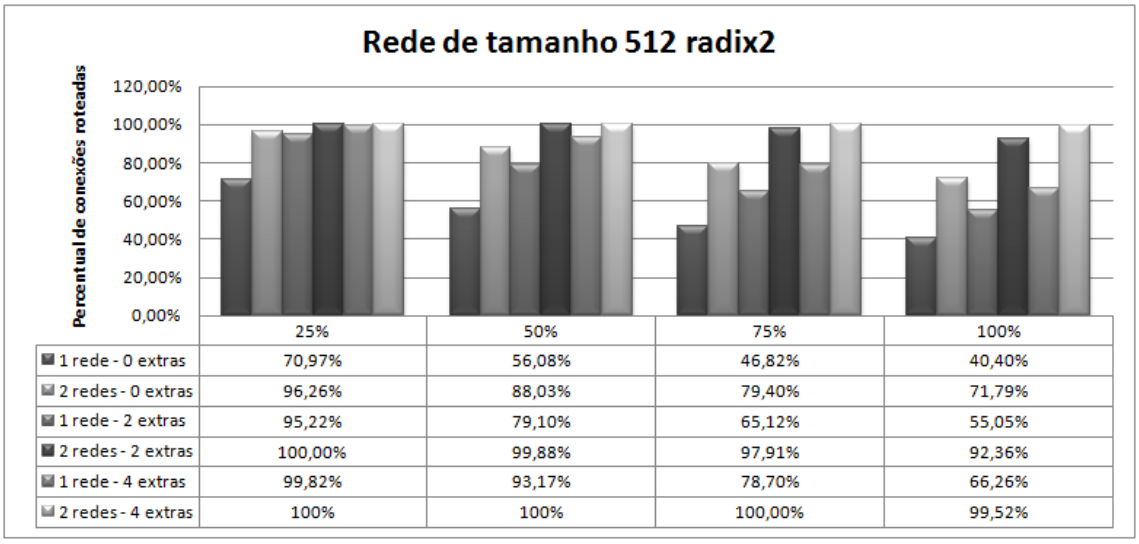
A Figura 4.15 apresenta os resultados para redes de tamanho 256. Neste gráfico, o pior resultado foi 43,07% das conexões roteadas com 1 rede e 0 estágios extras e com carga de trabalho 100%, podemos observar comparando com o gráfico da Figura 4.14, que quanto maior o número de entradas ou a rede, pior sua capacidade de roteamento em percentual de número de ligações. Com isso, diferente da rede de 64 onde todas as conexões puderam ser roteadas com 2 redes e 4 extras, aqui na rede de 256, com carga de trabalho de 100% não foi possível obter o 100% de roteamento, porém o valor ainda é alto, com um percentual de 99,88% de permutações realizadas com sucesso. As demais cargas de trabalho com duas redes e 4 estágios extras conseguiram rotear todas as conexões.



	25%	50%	75%	100%
1 rede - 0 extras	73,74%	59,02%	49,61%	43,07%
2 redes - 0 extras	97,30%	90,49%	82,76%	75,52%
1 rede - 2 extras	96,57%	82,29%	68,65%	58,42%
2 redes - 2 extras	100,00%	99,96%	98,96%	95,04%
1 rede - 4 extras	99,89%	95,07%	82,01%	69,76%
2 redes - 4 extras	100%	100%	100,00%	99,88%

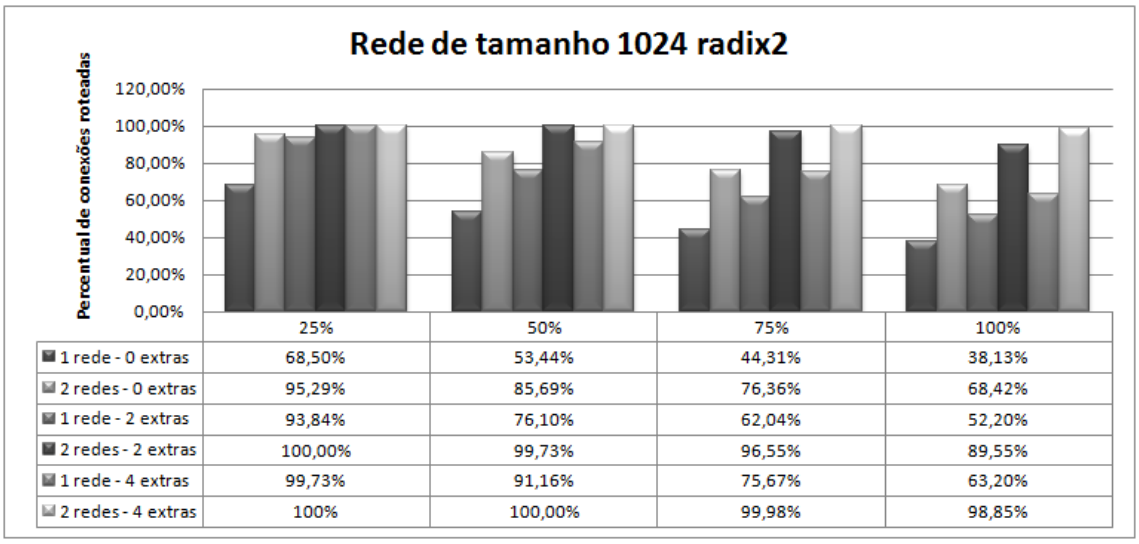
Figura 4.15: Roteamento rede de tamanho 256, com uma e duas redes, com e sem estágios extras.

A Figura 4.16 apresenta redes de tamanho 512 e a Figura 4.17 mostra redes de tamanho 1024. Quanto maior o tamanho da rede, menor a quantidade de conexões completamente roteadas. Para exemplificar melhor a rede de 512 como mostra a Figura 4.16, com 100% de carga com uma rede sem estágios extras, roteou 40,40% das conexões e a rede de 1024 conseguiu rotear 38,13%. Mesmo com a queda de capacidade de roteamento, uma rede de 512, ainda consegue rotear em 100% utilizando duas redes e 4 estágios extras em cargas de trabalho com até 75%. E uma rede de 1024 é possível rotear com sucesso todas as ligações com cargas abaixo de 50% de trabalho.



	25%	50%	75%	100%
1 rede - 0 extras	70,97%	56,08%	46,82%	40,40%
2 redes - 0 extras	96,26%	88,03%	79,40%	71,79%
1 rede - 2 extras	95,22%	79,10%	65,12%	55,05%
2 redes - 2 extras	100,00%	99,88%	97,91%	92,36%
1 rede - 4 extras	99,82%	93,17%	78,70%	66,26%
2 redes - 4 extras	100%	100%	100,00%	99,52%

Figura 4.16: Roteamento rede de tamanho 512, com uma e duas redes, com e sem estágios extras.

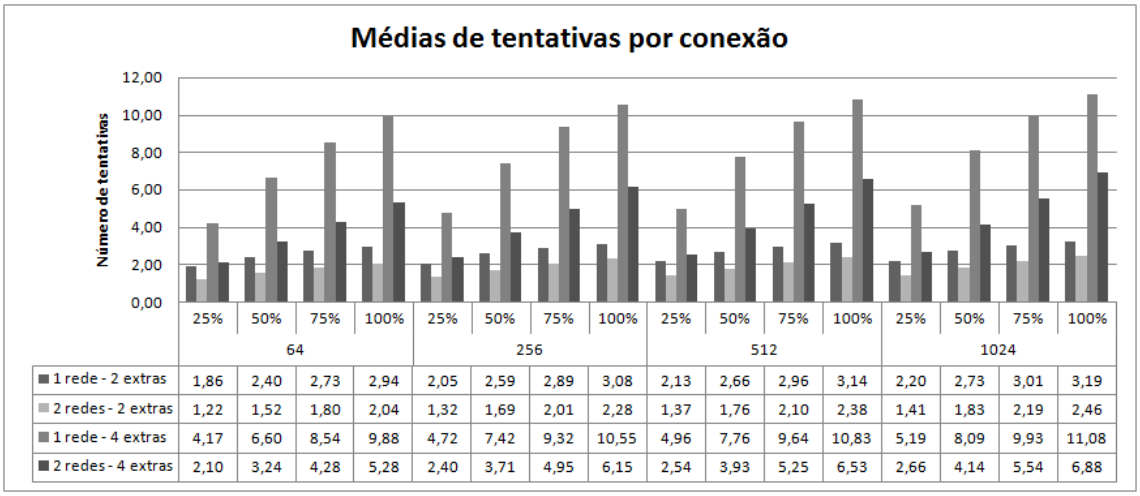


	25%	50%	75%	100%
1 rede - 0 extras	68,50%	53,44%	44,31%	38,13%
2 redes - 0 extras	95,29%	85,69%	76,36%	68,42%
1 rede - 2 extras	93,84%	76,10%	62,04%	52,20%
2 redes - 2 extras	100,00%	99,73%	96,55%	89,55%
1 rede - 4 extras	99,73%	91,16%	75,67%	63,20%
2 redes - 4 extras	100%	100,00%	99,98%	98,85%

Figura 4.17: Roteamento rede de tamanho 1024, com uma e duas redes, com e sem estágios extras.

4.5.2 Tentativas nos Estágios Extras Radix2

O segundo experimento contabiliza o número de tentativas que são avaliadas pelo algoritmo de roteamento em memória. No algoritmo com codificador de prioridade, todos as possibilidades são avaliadas em um único ciclo. Entretanto, para a versão baseada em memória, o tempo de execução irá depender da carga da rede e de sua configuração. Na Figura 4.18, podemos observar a média aritmética de tentativas por conexão para o código de estágios extras, vale ressaltar que para as redes sem extras, só existe um único caminho, então só há uma tentativa por conexão, o que muda quando inserimos estágios extras nas redes.



	64				256				512				1024			
	25%	50%	75%	100%	25%	50%	75%	100%	25%	50%	75%	100%	25%	50%	75%	100%
1 rede - 2 extras	1,86	2,40	2,73	2,94	2,05	2,59	2,89	3,08	2,13	2,66	2,96	3,14	2,20	2,73	3,01	3,19
2 redes - 2 extras	1,22	1,52	1,80	2,04	1,32	1,69	2,01	2,28	1,37	1,76	2,10	2,38	1,41	1,83	2,19	2,46
1 rede - 4 extras	4,17	6,60	8,54	9,88	4,72	7,42	9,32	10,55	4,96	7,76	9,64	10,83	5,19	8,09	9,93	11,08
2 redes - 4 extras	2,10	3,24	4,28	5,28	2,40	3,71	4,95	6,15	2,54	3,93	5,25	6,53	2,66	4,14	5,54	6,88

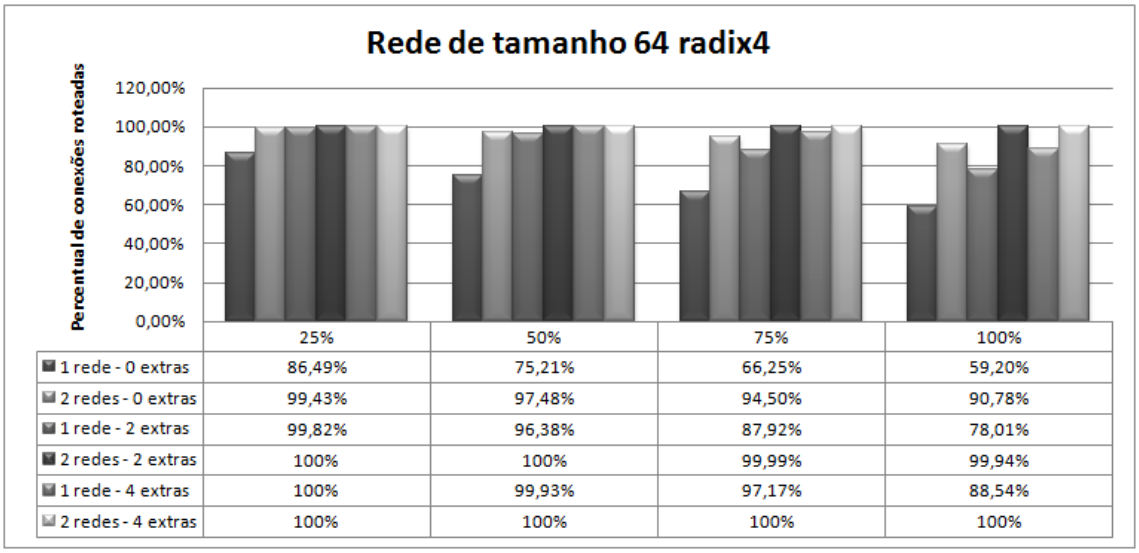
Figura 4.18: Médias de tentativas nas redes com estágios extras.

Como exemplo, vamos tomar um valor do gráfico da Figura 4.15, temos uma rede de 256 entradas, pegando uma carga com *50%* para uma rede e dois extras, teremos um percentual de conexões completas de *82,26%* e uma média de *2,59* tentativas, como pode ser observado na Figura 4.18. No pior caso seriam $2^2 = 4$, 2^n, mas o caso médio é *2,59*.

Algumas aplicações podem usar a rede para transmitir vários dados em vários passos. Nestes casos, uma rede com poucos estágios pode ser usada. Para aplicações que tem alta carga, a configuração com 4 estágios extras e duas redes tem bom desempenho. Por exemplo, para duas redes com 256 entradas e 4 estágios extras, todas as ligações são roteadas com sucesso com 75% de carga, ou seja, uma demanda de 192 ligações simultâneas. Apenas 4,9 tentativas por ligação, que significa uma média de 6 ciclos de relógio para rotear cada ligação. Para a carga de 1024 ligações, apenas 1,2%, ou seja, apenas 12 ligações não são incluídas e 1012 ligações são realizadas em paralelo com uma média de 6,8 tentativas por ligação com 2 redes e 4 extras. O pior caso seria 16 tentativas para 4 extras.

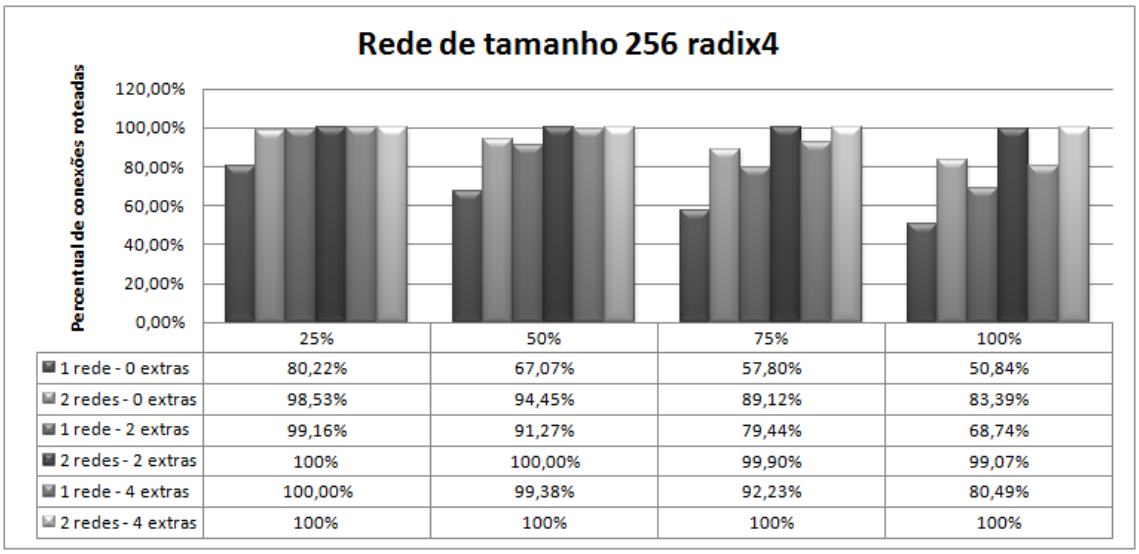
4.5.3 Capacidade de Roteamento Radix4

No terceiro experimento, repetimos a avaliação de capacidade de roteamento agora com radix4. O uso de radix4 irá reduzir os conflitos. As figuras, Figura 4.19, Figura 4.20, Figura 4.21, apresentam os resultados para uma e duas redes com radix4, ou seja, comutadores 4x4 no lugar de 2x2. A capacidade de roteamento melhora significativamente quando comparada as redes com radix2 das figuras, Figura 4.12, Figura 4.14, Figura 4.15. Para o caso da configuração com rede de tamanho 256, com 50% de carga e 2 estágios extras, temos cerca de 91% de ligações roteadas no lugar de 82% da radix2 de mesma configuração. Para as configurações com duas redes e 4 estágios extras (última linha da tabela), todos os roteamentos tem sucesso, exceto a configuração com 1024 entradas e permutação completa onde uma ligação em cada 16 permutações completas não é roteada com sucesso. As redes radix4 tem um custo menor que as radix2 nos FPGA Virtex6 com LUTs de 6 entradas como vimos nos resultados do Capítulo 3 e são uma alternativa a ser explorada.



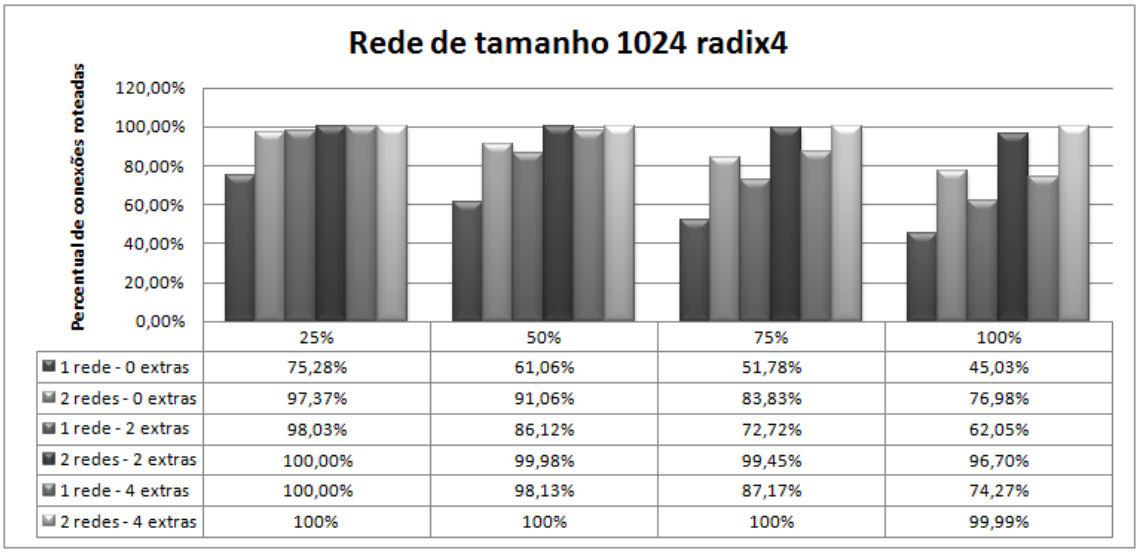
	25%	50%	75%	100%
1 rede - 0 extras	86,49%	75,21%	66,25%	59,20%
2 redes - 0 extras	99,43%	97,48%	94,50%	90,78%
1 rede - 2 extras	99,82%	96,38%	87,92%	78,01%
2 redes - 2 extras	100%	100%	99,99%	99,94%
1 rede - 4 extras	100%	99,93%	97,17%	88,54%
2 redes - 4 extras	100%	100%	100%	100%

Figura 4.19: Roteamento rede de tamanho 64, com uma e duas redes, com e sem estágios extras.



	25%	50%	75%	100%
1 rede - 0 extras	80,22%	67,07%	57,80%	50,84%
2 redes - 0 extras	98,53%	94,45%	89,12%	83,39%
1 rede - 2 extras	99,16%	91,27%	79,44%	68,74%
2 redes - 2 extras	100%	100,00%	99,90%	99,07%
1 rede - 4 extras	100,00%	99,38%	92,23%	80,49%
2 redes - 4 extras	100%	100%	100%	100%

Figura 4.20: Roteamento rede de tamanho 256, com uma e duas redes, com e sem estágios extras.

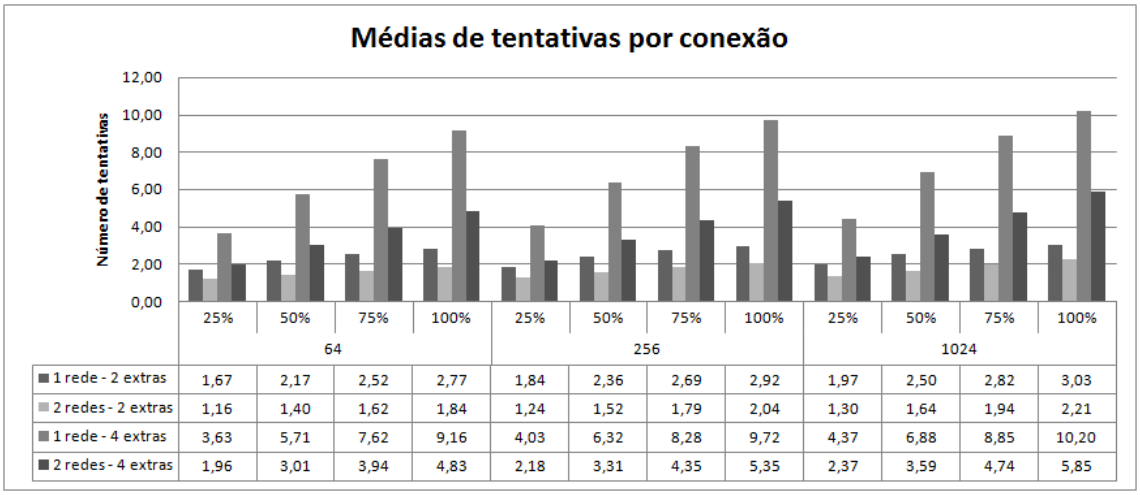


	25%	50%	75%	100%
1 rede - 0 extras	75,28%	61,06%	51,78%	45,03%
2 redes - 0 extras	97,37%	91,06%	83,83%	76,98%
1 rede - 2 extras	98,03%	86,12%	72,72%	62,05%
2 redes - 2 extras	100,00%	99,98%	99,45%	96,70%
1 rede - 4 extras	100,00%	98,13%	87,17%	74,27%
2 redes - 4 extras	100%	100%	100%	99,99%

Figura 4.21: Roteamento rede de tamanho 1024, com uma e duas redes, com e sem estágios extras.

4.5.4 Tentativas nos Estágios Extras Radix4

No quarto experimento, repetimos a análise do número de tentativas por conexão para o comutador radix4 no lugar do radix2. Na Figura 4.22, podemos observar a média aritmética de tentativas por conexão para o código de estágios extras.



	64				256				1024			
	25%	50%	75%	100%	25%	50%	75%	100%	25%	50%	75%	100%
1 rede - 2 extras	1,67	2,17	2,52	2,77	1,84	2,36	2,69	2,92	1,97	2,50	2,82	3,03
2 redes - 2 extras	1,16	1,40	1,62	1,84	1,24	1,52	1,79	2,04	1,30	1,64	1,94	2,21
1 rede - 4 extras	3,63	5,71	7,62	9,16	4,03	6,32	8,28	9,72	4,37	6,88	8,85	10,20
2 redes - 4 extras	1,96	3,01	3,94	4,83	2,18	3,31	4,35	5,35	2,37	3,59	4,74	5,85

Figura 4.22: Médias de tentativas nas redes radix4 com estágios extras.

A média de tentativas utilizando a rede radix4 também diminuiu. Comparando a mesma configuração feita na radix2 onde tínhamos uma média de 2,59 tentativas em uma rede de 256 entradas, pegando uma carga com *50%* para uma rede e dois extras, na radix4 teremos uma média de 2,36 tentativas.

Os experimentos anteriores mostraram que a rede oferece uma boa capacidade de roteamento e que o algoritmo em memória não necessita verificar todas as codificações em média. Os próximos experimentos irão avaliar os custos da implementação em FPGA dos algoritmos propostos aqui.

4.5.5 Implementação do Algoritmo de Roteamento em FPGA

O quinto experimento avalia a área ocupada e o atraso do algoritmo de roteamento baseado em memória. Uma descrição parametrizada em VHDL para a implementação do algoritmo de janelas foi realizada. A Tabela 4.1 apresenta os resultados da área ocupada na parte superior da tabela. O tempo de ciclo de relógio para efetuar o roteamento de uma ligação está na parte inferior da tabela. A síntese foi feita em uma FPGA Virtex6 usando o software ISE Xilinx 12.4. A ferramenta otimiza e implementa os módulos de memória com LUTs ou memórias embarcadas, dependendo do tamanho. Para redes com 64 e 256, a ferramenta optou por LUTs como mostra a Tabela 4.1 onde R2 e R4 são as versões radix2 e radix4, respectivamente. As linhas 0, 2 e 4 mostram as configurações com 0, 2 e 4 estágios extras. Apenas o algoritmo para 1024 entradas usa os módulos de memória embarcados. Uma Virtex6 tem 416 módulos e apenas 8 são usados na rede de 1024 radix2 com 4 estágios extras. Ou seja, a implementação consome um número bem reduzido de recursos do FPGA. Para as versões com LUTs, o consumo de recursos é menor ainda, em uma FPGA com 150.000 LUTs, se usa menos de 100 LUTs ou *0,1%* apenas do FPGA.

Estágios Extras	64(LUTs)		256 (LUTs)		1024(MEMs)	
	Radix2	Radix4	Radix2	Radix4	Radix2	Radix4
0	16	12	45	28	6	3
2	23	18	60	38	7	4
4	33	26	74	48	8	4
Ciclo de relógio (ns)						
0	1,5	1,5	1,8	1,8	3	2,6
2	1,5	1,8	1,8	1,8	3	2,7
4	3	3	3	3	3	3

Tabela 4.1: Área em LUTs/Memórias e Ciclo de relógio em ns para o Algoritmo de Janelas em Memória

Para o ciclo de relógio, apresentado na Tabela 4.1 na parte inferior, temos de 1,5 ns à 3,0 ns segundos. Além disso, o roteamento pode ser implementado em tempo de execução em

reconfiguração dinâmica. No pior caso, onde o algoritmo precisa fazer várias tentativas, o tempo para rotear uma ligação será de no máximo 10 a 11 ciclos como mostraram os experimentos 2 e 4 que avaliaram o número de tentativas por conexão. Ou seja, em apenas 30 ns é possível no pior caso, rotear uma conexão.

Comparado com uma implementação em software do algoritmo em uma máquina *dual core* com 2.8 Ghz utilizando apenas um núcleo, onde o roteamento de uma ligação gasta da ordem de 1 a 7 *micro*segundos, a aceleração do algoritmo paralelo em hardware é de duas à três ordens de grandeza em relação a versão sequencial em software.

O sexto experimento avalia os custos da implementação do algoritmo baseado no codificador de prioridade. Como veremos o custo é maior, porém em apenas 1 ciclo todas as possibilidades são verificadas. A Tabela 4.2 apresenta a área em LUTs e o ciclo de relógio para a implementação do roteamento com codificador de prioridade. Só são apresentados os resultados com estágios extras. A implementação sem estágio extra é direta, pois só existe um único caminho, não há necessidade do codificador de prioridade, que implica em um consumo significativamente menor em área e do ciclo de relógio. Pode-se observar que o consumo de LUTs é significativo, variando de *0,5% a 18%* de um FPGA Virtex6 em função do tamanho da rede e do número de estágios extras. Os resultados foram implementados com o codificador de prioridade parcial com *multicast*. Para uma rede de 64 entradas em radix4, usa-se *0,5%* do FPGA e o ciclo de relógio de 5 ns. A vantagem desta implementação é que sempre se gasta um ciclo de relógio para verificar o roteamento, enquanto que a versão com memória pode gastar de 3ns à 48ns, e em média gasta 30ns para um rede de 64 com 4 estágios extras. A versão com o codificador é, portanto 6 vezes mais rápida que a implementação com memória, apesar de ser 30 vezes maior. Para arquiteturas que tem disponibilidade de recursos nos FPGA e necessitam maximizar o desempenho, a implementação com codificador de prioridade é vantajosa.

Estágios Extras	64(LUTs)		256 (LUTs)		1024(LUTs)	
	Radix2	Radix4	Radix2	Radix4	Radix2	Radix4
2	1080	624	4951	2801	23020	13472
4	1313	729	5765	3316	26293	15106
Ciclo de relógio (ns)						
2	6,4	5,6	7,1	5,7	10,6	8,4
4	5,3	5,2	6,7	6,1	8,7	7,1

Tabela 4.2: Área em LUTs e Ciclo de relógio em ns para o Algoritmo com codificador de Prioridade

O codificador de prioridade é o elemento crítico. No sétimo experimento, um estudo do custo e atraso do codificador é realizado. A Tabela 4.3 apresenta os resultados da área ocupada em LUTs para três tamanhos de codificadores de prioridade, com 64, 256 e 1024 entradas. O número de entradas é mostrado na primeira linha. A segunda linha apresenta a área ocupada pelo codificador completo descrito em alto nível em VHDL e sintetizado pela ferramenta ISE da XILINX versão 12.4. A terceira e quarta linhas mostram o resultado da versão em 2 níveis, onde um codificador de 2^2 e de 2^4 é usado no segundo nível. Não há redução de área, apenas de atraso como ilustrado na Tabela 4.4 que mostra o atraso das versões de codificadores. O atraso depende da decomposição entre o primeiro e o segundo nível. A descrição do codificador também é em VHDL parametrizada e em alto nível. Finalmente na quinta e sexta linha são apresentados os resultados para o codificador parcial de 2 e 4 bits. A última linha mostra a redução de área do codificador parcial em relação ao em 2 níveis. A redução varia de 2 vezes a 4,6 vezes. Quanto maior o valor de *n* maior a redução, maior o ganho de desempenho. Para n = 1024, o ganho é 40 vezes em relação à descrição completa sem decomposição em alto nível.

N entradas		64	128	256	512	1024
completo		63	150	363	869	1690
2 níveis	2	87	183	388	687	1701
	4	82	146	421	876	1957
parcial	2	31	59	96	175	367
	4	42	60	123	320	368
redução		2	2,5	3	3,9	4,6

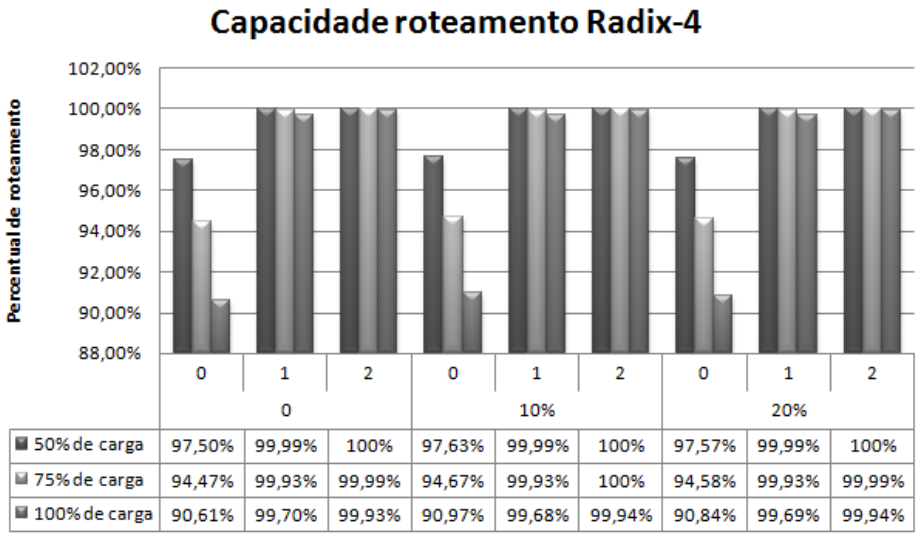
Tabela 4.3: Área em LUTs para Codificadores de Prioridade

N entradas		64	128	256	512	1024
completo		5,6	13	36	79	155
2 níveis	2	2,8	4,3	6,5	24	42
	4	5	3,5	3,5	4,8	7,6
parcial	2	2,9	3,15	3,2	3	3,7
	4	2,9	2,8	2,8	4	3,7

Tabela 4.4: Atraso em ns para Codificadores de Prioridade

4.5.6 Capacidade de Roteamento na Presença de Multicast

No oitavo experimento é avaliada a capacidade da rede em presença de padrões *multicast* de conexão. A rede avaliada tinha 64 entradas. As permutações parciais avaliadas tinham 50% e 75% das entradas/saídas ocupadas, o que equivale a 32 e 48 E/S respectivamente. Além destes testes, foi realizado um com permutações completas. Para cada experimento foram criadas aleatoriamente 100.000 permutações. A Figura 4.23 mostra os resultados. As permutações *tinham* de 10% e 20% de ligações com *multicast*, além de permutações sem *multicast*. Ou seja, para um carga de 50%, 4 das 32 conexões possuem *multicast*.



	0	1	2	0	1	2	0	1	2
	0			10%			20%		
50% de carga	97,50%	99,99%	100%	97,63%	99,99%	100%	97,57%	99,99%	100%
75% de carga	94,47%	99,93%	99,99%	94,67%	99,93%	100%	94,58%	99,93%	99,99%
100% de carga	90,61%	99,70%	99,93%	90,97%	99,68%	99,94%	90,84%	99,69%	99,94%

Figura 4.23: Análise roteamento conexões numa radix4 com e sem *multicast* para rede de tamanho 64.

Como podemos analisar, para instancias sem *multicast*, para 100.000 permutações completas, ou 6,4 milhões de conexões aleatórias, se não houver estágios extras, cerca de 90,71% de todas as conexões foram roteadas com sucesso. Entretanto se houver dois estágios extras, praticamente todas as conexões são roteadas com permutações completas. No caso de permutações parciais, apenas um estágio extra é o suficiente para rotear praticamente 100% das conexões. Como podemos analisar também, com *multicast* a rede não perde capacidade de roteamento, os resultados são parecidos com permutações sem *multicast*. Assim é possível utilizar a técnica de *multicast* sem diminuir a capacidade de roteamento da rede.

Como já mencionado, a execução do algoritmo de roteamento das redes multiestágio com estágios extras varia de acordo com o número de conexões. Para um e dois estágios extras, o número máximo de tentativas para cada conexão é respectivamente, 4 e 16. Porém como

podemos ver na Figura 4.24, a média de número de tentativas fica inferior a 2 para dois estágios extras. Portanto o algoritmo de roteamento com memória em hardware usará apenas 3 ciclos em média para rotear uma conexão.

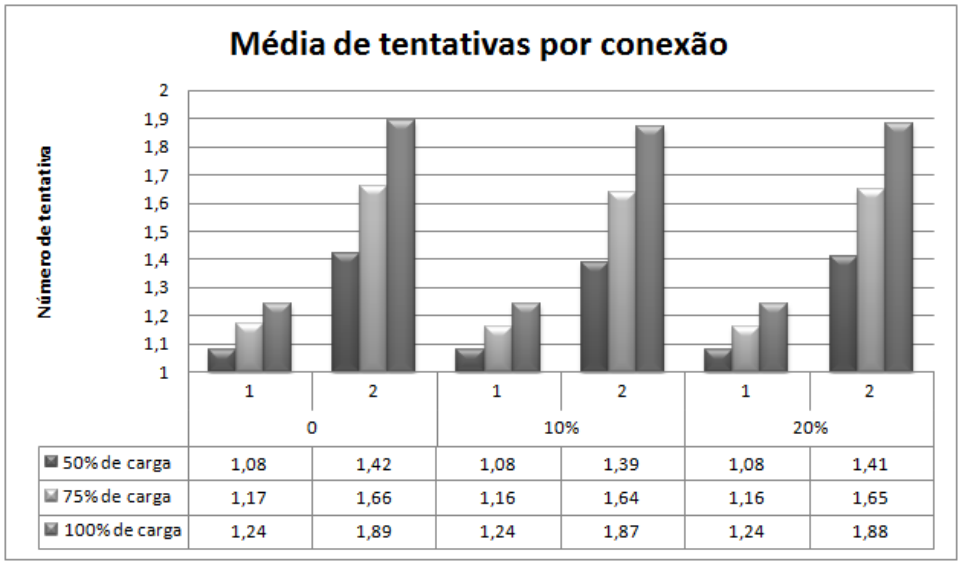


Figura 4.24: Médias de tentativas de roteamento por conexão, rede radix4 tamanho 64.

No nono experimento, a versão baseada em memória com *multicast* e radix4 foi implementada. A Tabela 4.5 mostra o número de recursos. Considerando uma rede de tamanho 64 com um estágio extra. Mapeadas em duas famílias de FPGA, Spartan3 e Virtex6. Para cada versão é avaliado duas opções de síntese, uma usando memória embarcada contidas no FPGA, e outra utilizando LUTs como memória. Em uma Virtex6 o custo é de apenas 0,03% que são 50 LUTs em cerca de 150.000 LUTs, sendo uma área muito pequena. E o algoritmo pode conseguir velocidades de até 200 Mhz. A primeira e terceira coluna de resultados da Tabela 4.5, mostra o resultado das arquiteturas para Sparta3 e Virtex6 sem utilizar memória embarcada, já a coluna 2 e 4 exibe os valores utilizando a arquitetura com memórias embarcadas.

Recursos	Spartan3		Virtex6	
Modulos de memórias embarcados	-	8	-	4
FlipFlops ou registros	30	6	28	4
LUTs de 4 entradas ou 6 entradas	165	32	50	26

Tabela 4.5: Recursos utilizados no FPGA para roteamento por hardware, rede de tamanho 64 com um estágio extra e *multicast*.

4.6 Considerações Finais

A avaliação do roteamento com carga de trabalho parcial mostra o potencial das redes. A maior parte dos trabalhos consideram permutações completas e radix2. Muitas aplicações fazem uso da maioria das ligações, porém, não necessariamente todas. Uma contribuição deste trabalho foi avaliar por amostragem a capacidade de uma e duas redes Omega com estágios extras em paralelo, além de avaliar o radix2 e radix4. Onde se mostrou que a utilização de estágios extras melhora a capacidade de roteamento das redes.

A maioria dos algoritmos de roteamento para redes multiestágio não foram validados em nível de tempo de execução. Este trabalho apresenta duas implementações paralelas do roteamento em hardware com avaliação de desempenho. As duas versões foram implementadas em FPGA. A versão com memória foi totalmente validada utilizando uma Virtex6. Esta versão teve o tempo médio de execução de 5 a 10 ciclos de relógio e usou poucos recursos do FPGA, menos de *1%*. Já a segunda versão baseada em codificador de prioridade, executa sempre em dois ciclos, um para verificar e outro para rotear, independente da carga da rede e do número de estágios extras. A área ocupada no FPGA pode ser significativa. Mas fica abaixo de *20%* para 1024 entradas em uma Virtex6 de 150.000 LUTs.

Foi proposto também um codificador de prioridade parcial, sintetizado em FPGA, mostrando uma redução em área de 4 vezes e uma redução do atraso em até duas vezes para as redes com 1024 entradas.

Por fim consideramos que o uso de redes Omega em FPGA é viável, principalmente para aplicações onde a carga de trabalho não é completa. E com o uso de algoritmos em hardware, podemos fazer o roteamento das redes em tempo de execução, sem ganho significativo do espaço interno do FPGA.

5. Algoritmo Escalonamento, Posicionamento e Roteamento em uma Arquitetura CGRA com Multiestágio

Este Capítulo apresenta uma aplicação das redes multiestágio em arquiteturas de grão grosso [Ferreira, 2011a]. As arquiteturas reconfiguráveis de grão grosso (CGRA - *Coarse-Grained Reconfigurable Architecture*) são um modelo promissor para sistemas embarcados, pois são uma solução que reduz a complexidade de síntese e mapeamento no FPGA. Consequentemente reduzem o tempo de reconfiguração. Apesar destas vantagens, existem poucos CGRA no mercado [Friedman, 2009] [Mei, 2003] [Baumgarten, 2003] e poucas ferramentas comerciais para o mapeamento das aplicações. Assim este trabalho propõe implementar um CGRA virtual em um FPGA comercial como uma camada virtual de reconfiguração. A arquitetura proposta é composta por Unidades Funcionais heterogêneas (FU) e por uma rede de intercomunicação multiestágio para interligar seus componentes. Além da arquitetura, também é apresentado um algoritmo de escalonamento e mapeamento de aplicações a partir de grafos de operadores com fluxo de dados. O uso da rede multiestágio como um mecanismo de comunicação global, gera uma redução significativa do esforço para o posicionamento e o roteamento. Tendo como base esta arquitetura, este trabalho apresenta também uma heurística com complexidade polinomial de escalonamento, posicionamento e roteamento (SPR).

Inicialmente, a arquitetura com o mecanismo de comunicação global é apresentada. Em seguida, um exemplo de mapeamento de um grafo com operadores na arquitetura é apresentado. Posteriormente, o escalonamento é exemplificado em alto nível e o conceito de intervalo de inicialização (II) e partição temporal é introduzido. Então, o algoritmo de escalonamento, posicionamento e roteamento é apresentado na Seção 5.4. Diferente de outros autores [Mei, 2003] [Micheli, 1994], onde os três passos são implementados em separado, a abordagem proposta aqui é uma heurística que integra os três passos. O escalonamento com um determinado intervalo II pode falhar caso não seja possível realizar o posicionamento ou roteamento. Nas Seções 5.5 e 5.6, estes aspectos são discutidos.

5.1 Arquitetura reconfigurável proposta

Será abordada nesta sessão uma arquitetura CGRA com redes multiestágios radix4 para interligar as unidades de processamento. Um dos nossos objetivos é avaliar o uso das redes multiestágio para implementação deste tipo de problema.

A Figura 5.1 apresenta a arquitetura composta por unidades funcionais (*function units - FUs*), registradores e uma rede global de interconexão. As FUs podem ser, somadores, multiplicadores, unidades lógicas, dentre outras, ou podem ser unidades que fazem vários tipos de processamento. A rede de interconexão global é formada por duas redes multiestágios (*MINs*) radix4. Cada FU é conectado as duas redes multiestágio. As saídas da rede de cima são conectadas na primeira entrada de cada *FU*. A rede de baixo é conectada na segunda entrada de cada *FU.* As entradas e saídas externas são conectadas em ambas as redes. A rede global permite conectar diretamente qualquer par de FU. Para conectar a FU 0 na FU 1, basta solicitar o roteamento de uma conexão da entrada 0 da rede para a saída 1.

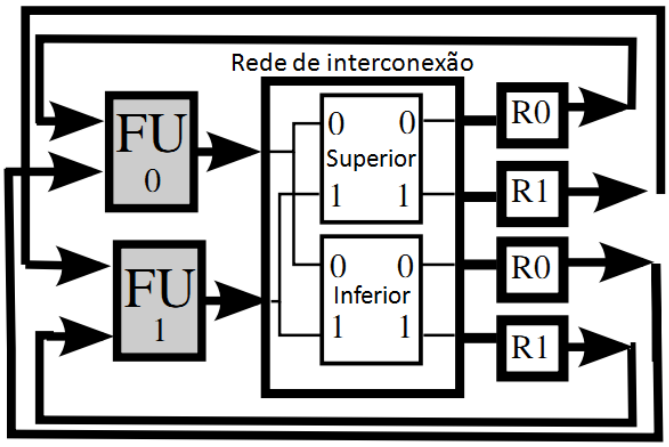


Figura 5.1: Arquitetura reconfigurável proposta

5.2 Exemplo de Mapeamento

Será apresentado um exemplo usando a arquitetura proposta, mapeando uma aplicação simples modelada com um grafo de operadores. A Figura 5.2(a) mostra o grafo que será mapeado, onde os nós representam as operações e as arestas, as relações de dependências entre as operações. Vamos supor uma arquitetura que possui apenas duas FUs. Uma possibilidade para o mapeamento é usar três configurações na rede de interconexão. Cada configuração será executada em um ciclo. Começando a execução, os nós A e B serão colocados na primeira configuração como mostra a Figura 5.2(b). Depois da execução, os

resultados são roteados através da rede de interconexão global para a próxima configuração. Como é mostrado na Figura 5.2(c), os valores de A e B são armazenados nos registros. O valor de B está em dois registros porque ele é necessário para o cálculo tanto de C quanto de D, assim foi necessário alocá-lo em um registro que alimenta a FU0 e um registro que alimenta a FU1. No segundo ciclo de operação os nós C e D são posicionados, os resultados são computados e enviados para os registros através da rede de interconexão para serem usados no próximo ciclo. Finalmente, a operação E é comutada no terceiro ciclo usando os valores de C e D lidos dos registros.

Esta arquitetura é flexível e várias estratégias de escalonamento e posicionamento podem ser implementadas. Abordaremos neste capítulo uma técnica de escalonamento utilizando *pipeline* e particionamento temporal em módulo *scheduling*.

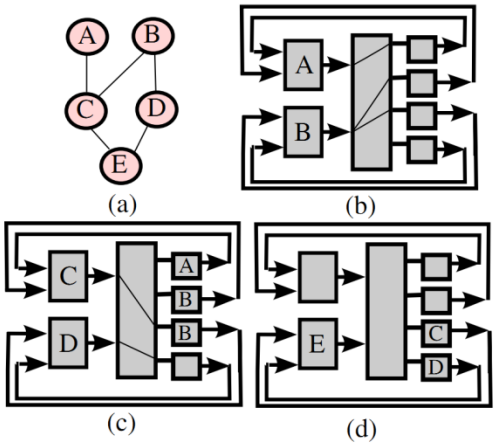


Figura 5.2: Um exemplo simples de mapeamento.

5.3 Escalonamento

Foi proposto neste trabalho uma heurística com complexidade polinomial para fazer o escalonamento. O objetivo é incorporar este algoritmo em um ambiente de compilação *Just-in-time* para mapeamento de aplicações com grande paralelismo de instruções em CGRA implementados em FPGA comerciais.

Primeiro, os algoritmos ASAP e ALAP são executados no grafo para determinar a flexibilidade de escalonamento de cada operação. No segundo passo, registros são inseridos entre os vértices cujos caminhos de dados estão desbalanceados. Após isso, o posicionamento e o roteamento são feitos em apenas um passo. A implementação opera em *pipeline*. Quando o

grafo é menor que a arquitetura, ou seja, o número de vértices é menor que o total de FU, todo o grafo pode ser implementado em uma única configuração.

Entretanto, semelhante a trabalhos anteriores [Mei, 2003], se o grafo é maior que a arquitetura, é necessário particionar em configurações e os dados são inseridos em intervalos. O intervalo de inicialização (II) é inicializado com o menor valor possível. Por exemplo, se temos 10 vértices e 5 FU, então o valor mínimo será 10/5= 2. Se não for possível fazer o mapeamento, o valor de II é incrementado e uma nova tentativa de mapeamento é realizada. Este procedimento é repetido até todo o grafo conseguir ser mapeado na arquitetura.

Para facilitar o entendimento, vamos considerar novamente o exemplo do grafo da Figura 5.2, porém com algumas considerações adicionais para a arquitetura. Vamos supor agora que a arquitetura possua três FUs, como mostra a Figura 5.3. Além disso, sinais externos são incluídos para inserir dados nos nós A e B. Como o número de nós é maior que o número de FUs, pelo menos duas configurações serão necessárias. E o intervalo de inicialização(II) mínimo será 2. A Figura 5.3(a) mostra o grafo de dados. No ciclo T_i, os nós A e B são computados e colocados no segundo e terceiro FU (veja Figura 5.3(b)). Os resultados são roteados através da rede de interconexão para serem usados no próximo ciclo. No ciclo T_{i+1}, os nós C e D recebem os dados de A e B como entradas no registros, e o resultado é roteado através da rede de interconexão para o primeiro FU, onde o nó E será colocado no ciclo T_i. Assim os nós A, B e E serão colocados no tempo T_i, e os nós C e D serão colocados na configuração T_{i+1}. Ou seja, temos duas partições ou configurações temporais T_i e T_{i+1}. É importante lembrar que os nós que compartilham a mesma configuração, como A e E, podem ter dados de intervalos diferentes. Enquanto E está no Tempo T_0, C e D estão no tempo T_1 e, A e B estão no tempo T_2.

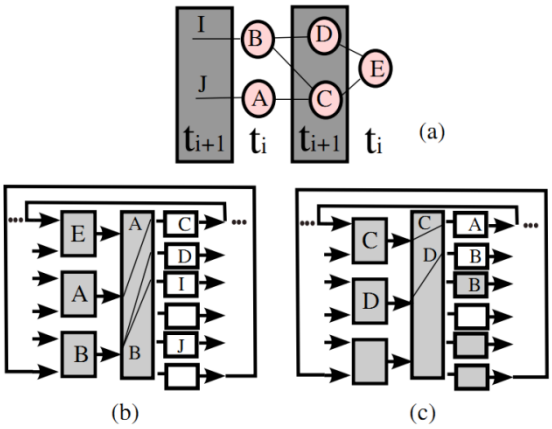


Figura 5.3: Um exemplo utilizando *pipeline*.

A latência de execução é de quatro ciclos, porém a vazão de dados será de dois ciclos, que é igual ao II. No caso de arquiteturas pequenas, mais configurações se fazem necessárias. Supondo o exemplo anterior mapeado em uma arquitetura com duas unidades, como ilustrado na Figura 5.4. Adicionalmente esta arquitetura possui um registro. Como não é possível mapear A e B e E na mesma configuração, o escalonamento insere um registro deslocando o nó. Assumindo que o nó A é deslocado. Os nós A e E são computados na mesma configuração T_i, os nós C e D são computados na configuração T_{i+2}, e o registro que repassa o valor de A e o nó B são computados na configuração T_{i+1}(veja Figura 5.4). Com isso a latência vai para 5 ciclos, e a vazão dos dados vai para 3 ciclos. Isto irá ocorrer se o grafo for muito maior que a arquitetura.

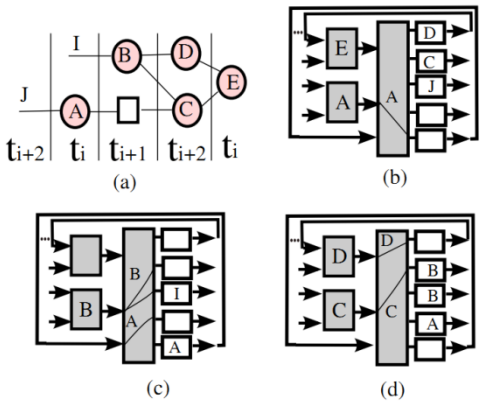


Figura 5.4: Exemplo de utilização de registro para passagem de valor do nó.

5.4 Algoritmo Escalonamento, Posicionamento e Roteamento (SPR)

O pseudo código do algoritmo SPR (*Schedulead Placement Route*) é mostrado na Figura 5.5. Primeiro o escalonamento usando ASAP e o ALAP é computado. Depois o II mínimo é calculado com base no número de FUs disponíveis na arquitetura e exigidos no grafo. A primeira configuração é inicializada. O mapeamento é computado nas linhas 5 até a 19. Das saídas para as entradas. O grafo é percorrido usando níveis ALAP. Cada nível é mapeado na configuração atual. Na linha 10, um nó *n* é removido do nível atual. Então o posicionamento é verificado. Se não é possível fazer o posicionamento ou o roteamento, o valor de II é aumentando em 1 e um novo mapeamento é computado. Como o número de configurações aumenta o número de recursos requeridos por configuração diminui, então o mapeamento com o novo valor de II poderá ser possível de ser realizado.

```
Inputs: grafo G, Arquitetura A
1  Asap_Alap(G);
2  II = menor_recurso(G,A);
3  mapeado = false;
4  Cfg = II.obtem_primeiro();
5  While ( ! mapeado ) {
6  For cada nível L in G das saidas do
7  {
8   while ( L.nao_vazio() )
9   {
10     n = L.remove_no();
11     if ( ! posicionamento(n,Cfg)  )
12     falhou;
13  }
14  if (falou) break;
15  Cfg = II.obtem_proximo();
16 }
17  if (falhou) Cfg = II.incrementa();
18  else mapeado= true;
19 } // mapeamento
20
21  posicionamento (no n, Config Cfg) {
21      FU = Cfg.obtem_FU(n);
22      if ( FU = null ) { // FU Perdida
23      FU = Cfg.obtem_registro();
24      if ( FU = null ) return false;
25      reescalona(n);
26      }
27      return route(n,FU,Cfg);
28  } // fim posicionamento
29
30  rotemaneto(no n, unid FU, Config Cfg) {
31      saida = n.obtem_saida();
32      FU_saida = saida.obtem_FU();
33      sucesso = roteamento_rede(FU,FU_saida);
34      if (! sucesso ) {
35          FU = Cfg.obtem_registro();
36          if ( FU = null ) return false;
37          reescalona(n);
38          n.set_FU(FU);
39          sucesso =
roteamento_rede(FU,FU_saida);
40      }
41      return sucesso;
42  } // fim roteamento
```

Figura 5.5: Pseudo código do algoritmo de escalonamento, posicionamento e roteamento(SPR).

A seguir, um exemplo será usado para explicar os conceitos de II e escalonamento em um exemplo maior. A Figura 5.6 mostra a grafo representando o fluxo de dados extraídos do algoritmo *Motion Vector* retirado do *Express Benchmark* disponível em [ExPRESS Benchmarks] e duas configurações de escalonamento. Supondo que o valor inicial de II é 2. A primeira configuração C_0, ordenada pelo nível ALAP, possui 7 somadores, 7 multiplicadores, e 2 operadores de *load*. A segunda configuração possui valores similares, 7 somadores, 7 multiplicadores, e dois operadores de *store*. Como podemos observar na Figura 5.6(a). O número máximo de somadores e multiplicadores é 7. Neste exemplo, estamos supondo uma arquitetura heterogênea, onde as FU tem funções especificas. Se II é incrementado para três, o número de recursos por configuração diminui. A primeira configuração C_0 terá 5 somadores, 4 multiplicadores, e dois operadores de *store*. A segunda configuração C_1 terá 5 somadores, e 4 multiplicadores, e dois operadores de *store*, e a configuração C_2 terá 4 somadores, e 5 multiplicadores. O número máximo de somadores e multiplicadores será 5 por configuração. Como mostra a Figura 5.6 para três configurações ou II = 3.

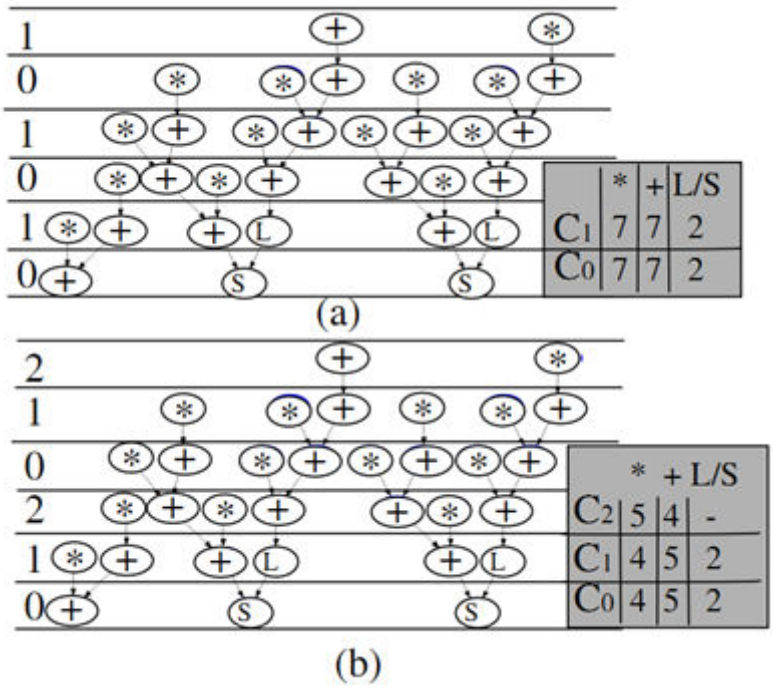


Figura 5.6: Dois escalonamentos do grafo *Motion Vector*. (a) com II igual a 2 (b) com II igual a 3

5.5 Posicionamento

O posicionamento é direto, uma vez que a arquitetura possui um mecanismo de comunicação global, um vértice pode ser posicionado em qualquer FU que esteja disponível. No caso de

uma arquitetura heterogênea, onde as FU possuem funções especificas, o vértice deve ser posicionado em qualquer FU que execute a operação requisitada pelo vértice. O exemplo anterior mostrou um caso onde o número de operadores são bem distribuídos e balanceados entre as configurações, não ocorrendo problemas de posicionamento. Agora, considerando o grafo de fluxo de dados do "*finite pulse filter*" (FIR) [ExPRESS Benchmarks], mostrado na Figura 5.7. Vamos supor que a arquitetura tenha somente 8 somadores, 6 multiplicadores e 12 entradas/saídas. O grafo possui, 10 somadores, 11 multiplicadores e 23 entradas/saídas. Assim pelo menos 2 configurações são necessárias. Para duas configurações, são 2*8 = 16 somadores, 2/6 = 12 multiplicadores e 2*12=24 entradas/saídas. Utilizando a ordem do ALAP sem reescalonar, C_0 precisará de 19 entradas/saídas e C_1 precisará de 7 multiplicadores, como podemos ver na Figura 5.7(a). Ou seja, não é possível posicionar 7 multiplicações em 6 FU com a função de multiplicação.

O algoritmo de SPR começa das saídas para as entradas. O escalonamento é feito para um determinado II. Para verificar se é possível escalonar, o posicionamento e roteamento do grafo na arquitetura escolhida são executados para cada nó até não haver mais FUs (falha de posicionamento) ou um conflito de roteamento ocorrer.

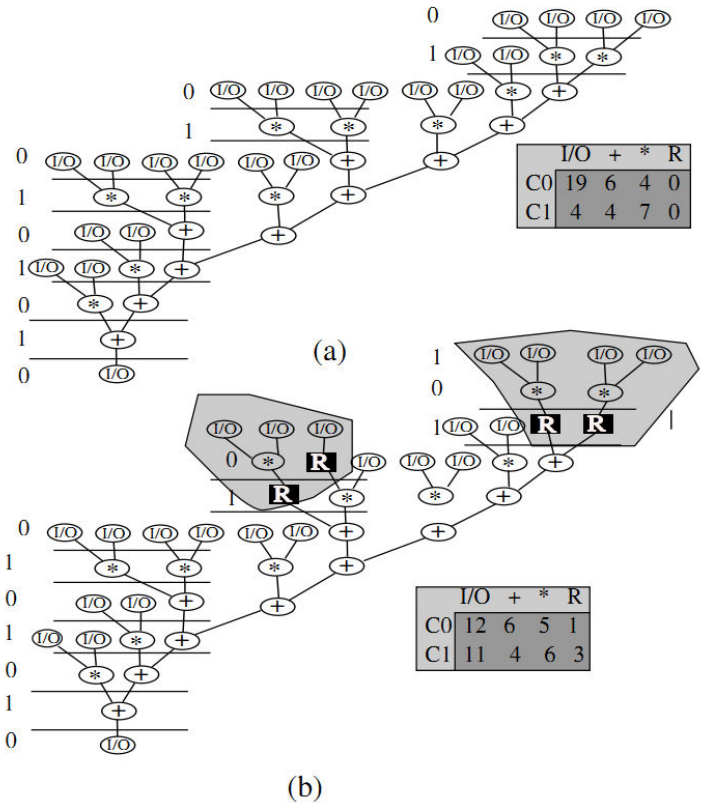


Figura 5.7: Grafo FIR; (a) Ordem do ALAP; (b) SPR

O algoritmo de posicionamento é descrito nas linhas 21-29 da Figura 5.5. Na Linha 21 a função Cfg.obtem_FU retorna um FU livre para cada operação. Se não houver mais FU, o posicionamento colocará um registro. O registro irá deslocar a operação de uma partição temporal para outra. No exemplo anterior, ao inserir o sétimo multiplicador na configuração 1, ocorrerá uma falha de posicionamento por falta de unidade, como mostra a Figura 5.7(b). Um registro será inserido e roteado na configuração 1, e a multiplicação será escalonada para a configuração 0, com também as suas entradas descendentes. O Algoritmo SPR continuará até mais dois multiplicadores sejam requisitados para a configuração 1, e dois registros sejam inseridos e as operações serão movidas para a configuração 0. Como consequência indireta quando os multiplicadores são movidos para a configuração 0, os operadores de entrada/saída são movidos para a configuração 1. Quatros registros são incluídos e a latência aumenta em um ciclo. Porém a vazão dos dados é de dois ciclos. Além disso, existe o alto grau de paralelismo do grafo capturado pelo SPR, pois, praticamente todas as FUs são usadas em todos os ciclos, mesmo em uma arquitetura com poucos recursos.

5.6 Roteamento

O roteamento é integrado no algoritmo de escalonamento e posicionamento proposto no SPR. O pseudo código é descrito nas linhas 30-41 da Figura 5.5. O escalonamento e posicionamento varrem das saídas para as entradas. Quando uma FU é posicionada, precisamos de rotear a saída da FU. A função roteamento_rede tenta rotear o FU da atual configuração com a FU colocada na configuração anterior. Para facilitar a explicação, o código da função não foi detalhado. Se houver um conflito para a FU atual, outra FU compatível será testada. Se nenhuma FU compatível pode ser conectada com a FU de saída, o roteamento falha. Nesse caso o algoritmo tenta inserir um registro (linha 34-40). Se não for possível rotear nenhum registro ou não houver mais registros livres, o mapeamento falha, e o II é incrementado. A rede de interconexão utiliza duas redes radix4 em paralelo. No roteamento, tenta-se utilizar a primeira rede, se ela não estiver disponível o roteamento tenta ser feito na segunda rede. Como já falado, o algoritmo de roteamento da rede radix4 é rápido e possui complexidade O(Lg n), então uma das vantagens é a rápida velocidade de execução do algoritmo, podendo assim ser integrado em compiladores *Just-in-time* ou mesmo em sistemas de reconfiguração dinâmica.

5.7 Resultados Experimentais

A arquitetura e o algoritmo SPR foram avaliados com um subconjunto de grafos de fluxos de dados extraídos do conjunto *Express Benchmark* disponível em [ExPRESS Benchmarks]. Este subconjunto de grafos foram selecionado entre mais de 1400 grafos de fluxo de dados de aplicações do Mediabench *benchmark* [ExPRESS Benchmarks]. As características dos grafos em função do número e tipo de operações são apresentadas na Tabela 5.1. Colunas E/S, Reg, M, +, *, L mostram os números de entradas/saídas, registros, operações com memória, somadores, multiplicadores e operadores lógicos. Como alguns grafos disponíveis em [ExPRESS Benchmarks] possuem nós de somadores e multiplicadores sem nenhuma entrada, nós decidimos incluir duas entradas externas nesses nós. A coluna Reg mostra o número de registros adicionados para balancear as arestas com reconvergências desbalanceadas para a execução em *pipeline*.

Nome	E/S	Reg	M	+	*	L
collapse	12	40	18	29	9	12
feedback	42		11	23	18	6
fir	23			10	11	
fir1	17			15	8	
h2v2	36	10	17	32	3	4
Inter	96		16	56	36	4
matmul	50		24	45	40	5
invert	154	24	80	106	141	22
jpeg slow	52	100	24	78	37	16
jpeg fast	54	253	24	78	37	16
smooth	130	8	48	80	69	9
writebmp	76	6	35	37	2	57

Tabela 5.1: Características do grafos utilizados

A Tabela 5.2 mostra seis configurações de arquiteturas que foram avaliadas. Estas arquiteturas foram escolhidas para avaliar o SPR considerando diferentes números de unidades funcionais. Portas de E/S, somadores, multiplicadores, operadores lógicos, *load/stores* e registros. Pois cada um destes elementos pode influenciar os resultados do SPR. A primeira coluna mostra o número para identificar a arquitetura. A coluna *tam* mostra o número de entradas da rede de interconexão. Nós avaliamos três redes de tamanho médio com 64 entradas, e três redes de

tamanho grande com 256 entradas. A redes foram implementadas com radix4 reduzindo a latência e o tamanho em um FPGA Virtex6.

Nome	tam	+	*	L	M	E/S	R	Total
A1	64	10	10	5	5	16	18	30
A2	64	18	8	4	4	12	18	34
A3	64	10	8	4	4	20	18	26
A4	256	48	48	28	28	64	40	152
A5	256	60	32	26	26	72	40	144
A6	256	48	32	20	20	96	40	120

Tabela 5.2: Características das arquiteturas escolhidas

A última coluna mostra o número total de FU, sem levar em consideração o número de registros e de E/S. A arquitetura A_2 tem mais somadores que a A_1, e a arquitetura A_3 tem mais E/S. Para a arquitetura A_5, o número de somadores é maior como também o número de E/S que a arquitetura A_4. A arquitetura A_6 tem mais E/S que a A_4. Este trabalho é apenas uma exploração inicial para validar o algoritmo SPR e verificar o tempo de execução. Uma exploração mais amplas das arquiteturas pode ser realizada em um estudo mais aprofundado do mapeamento de um conjunto mais amplo de aplicações.

A Tabela 5.3 mostra os resultados do SPR para um subconjunto de grafos mapeados na arquitetura A_1 detalhada na Tabela 5.2. A primeira coluna mostra o nome do grafo. A segunda contém o número de nós sem levar em consideração os registros. A latência mínima e o II mínimo são mostrados nas colunas Latência e II Mínimo. O II mínimo é calculado baseado nos recursos de FU. O número de instruções por ciclo é mostrado na coluna IPC, que mostra quantas operações são computadas em média por ciclo. Se nós dividirmos o IPC pelo número de FU sem levar em conta os registros, teremos uma densidade média de cerca de 50% usados em recursos. Isto mostra a média de números de FU ativos por ciclo. O IPC é alto, variando de 19 até 26. O algoritmo SPR proposto é uma heurística gulosa e em média o valor do II foi incrementado em 20% a mais que o mínimo valor de II. O algoritmo SPR tem um tempo de execução baixo. O tempo de execução é mostrado na última coluna. Os testes foram feitos em um Intel Core 2 Duo P8400 de 2.26 Ghz. A média de execução é de 40 milissegundos, mostrando a viabilidade de sua implementação em compiladores *Just-in-Time*.

Nome	Número Nós	Latência	II mínimo	IPC	II	Tempo Execução(ms)
collapse	80	10	3	19,25	4	17,63
feedback	100	9	3	25	4	23,71
fir	45	12	2	22	2	14,04
fir1	40	11	2	20	2	10,29
h2v2	90	20	4	22,5	4	49,92
Inter	208	10	6	26	8	117,32
matmul	164	11	5	23,43	7	63,18

Tabela 5.3: Resultados do SPR para a arquitetura A_1

A Tabela 5.4 mostra o resultado do algoritmo SPR com um subconjunto de grafos na arquitetura A_6. Esta tabela é similar a Tabela 5.3. Entretanto grafos maiores foram avaliados. Grafos maiores não são mapeados na arquitetura A_1 devido à falta de registros. A arquitetura A6 é significativamente maior que a A1. A A6 possui apenas 40 registros, porém unidades de soma podem funcionar como registros. O aumento médio do II foi de 40%. O IPC obtido é alto, variando de 37 até 104. O resultado mostra um valor médio de 4 para o II. O tempo médio de execução foi de 130 milissegundos.

Nome	Número Nós	Latência	II mínimo	IPC	II	Tempo Execução(ms)
inter	208	10	2	104	2	51,71
invert	503	13	5	84	11	59,42
jpeg slow	196	17	3	65,33	3	180,05
jpeg fast	187	20	4	37	5	336,25
matmul	164	11	1	82	2	59,25
smooth	336	13	3	67,2	5	180,22
writebmp	207	9	2	103	2	69,49

Tabela 5.4: Resultados do SPR para a arquitetura A_6

A Tabela 5.5 mostra o total de recursos utilizados no FPGA para cada arquitetura da Tabela 5.2. Os resultados foram gerados com a ferramenta ISE 12.4 da Xilinx para uma arquitetura Virtex6. As arquiteturas pequenas com redes de tamanho 64 usaram em média 1% dos registros do FPGA, 15% das LUTs do FPGA e 4% dos DPS. Os DSPs são usados para implementar os multiplicadores. A maior arquitetura usa 6% dos recursos de registros, 82% das LUTs e 25% dos DPSs do FPGA. Nestas arquiteturas não foram utilizadas memórias embarcadas (BRAMs). A área ocupada pelo FPGA é dominada pela rede de comunicação global. Porém é importante

destacar que se fossem usadas redes *crossbar*s ao invés das multiestágios, a rede A1 ocuparia 80% do FPGA, e não seria possível implementá-la.

	Slices	Slices		Clock
arch	Registers	LUTs	DSP	MHz
A1	4864	22609	40	100
A2	4701	22545	32	100
A3	4991	22731	32	100
A4	18625	123958	192	80
A5	18881	124230	128	80
A6	19700	123457	128	80

Tabela 5.5: Arquiteturas mapeadas em FPGA, Xilinx Virtex6.

A Tabela 5.6 mostra os valores comparativos do tamanho e velocidade da arquitetura proposta comparando-a com os grafos aqui analisados mapeados diretamente no FPGA. O mapeamento direto seria implementar o grafo como uma estrutura fixa, com todas as unidades necessárias implementadas espacialmente no FPGA.

Aplicação	Ganho Total	Ganho do PAR	Overhead da Área do CGRA	Overhead do Clock do CGRA
collapse	11.798	5.842	20	3
feedback	8.899	4.175	26	3
fir1	17.687	9.135	60	3
fir	13.177	6.766	70	5
h2v2	4.107	2.083	20	2
inter	4.371	2.050	67	3
invert	6.075	2.255	29	2
jpeg_slow	1.466	628	51	3
jpeg_fast	812	336	51	4
matmul	3.831	1.755	83	3
Média	7.222	3.503	48	3

Tabela 5.6: Comparativo, entre tempo de síntese, posicionamento e Roteamento, Área Ocupada, e Frequência de operação.

Como se pode observar, o tempo total de escalonamento, posicionamento e roteamento na arquitetura, foi em média 7.222 vezes mais rápido que a arquitetura sendo sintetiza e mapeada diretamente no FPGA. Mesmo contando somente o tempo de posicionamento e Roteamento no FPGA, esse ganho continuou sendo de 3.503 vezes. Por outro lado, a

arquitetura ficou maior, devido às redes de interconexão que não existem no mapeamento direto. O aumento ficou em média 48 vezes maior. O *clock* também diminui já que a arquitetura proposta precisa percorrer um caminho maior possuindo as redes de interconexão. Em média a frequência ficou 3 vezes menor. Mas o tempo de execução do SPR mostra que nossa arquitetura possui um escalonamento, posicionamento e roteamento muito mais simples como se pode observar na Tabela 5.6. Tornando nossa arquitetura uma solução para aplicações onde são executados vários aplicativos diferentes, e os resultados devem ser obtidos *Just-in-time*.

5.8 Considerações Finais

Foi proposta neste capítulo uma arquitetura reconfigurável de grão-grosso, utilizando redes de interconexões multiestágio fazendo as interconexões globais. Além de um algoritmo de escalonamento, posicionamento e roteamento (SPR) usando *pipeline* para esta arquitetura. Os resultados experimentais mostraram que cerca de 50% dos recursos da arquitetura foram utilizados em média por ciclo de *clock*. Diferente dos trabalhos com CGRA que demandam a construção física das arquiteturas [Volker, 2003] onde seriam necessárias a produção de novos *chips* com tal arquitetura, a arquitetura proposta aqui é implementada virtualmente em uma arquitetura de FPGA comercial.

Mostrou-se que o uso de redes radix4 é viável, gerando uma redução significativa em comparação com redes *crossbar*. Além disso, o algoritmo de roteamento possui complexidade O(N Lg N), integrado ao SPR gera um tempo de execução baixo que possibilita sua integração em compiladores JIT *(Just in Time)*.

6. Redes reguladoras de genes em FPGA

Este capítulo apresenta uma aplicação de bioinformática implmentada em hardware reconfigurável com o uso das redes de interconexão. Os resultados demonstram um ganho de desempenho de 2 a 3 ordens de grandeza de desempenho em relação à solução em software. A aplicação é modelada como um grafo Booleano e implementada em FPGA. O grafo representa uma rede reguladora de genes. A modelagem de redes de genes é usada em aplicações de biologia molecular, como a compreensão dos processos metabólicos no interior de uma célula, a busca por medicamentos, os estudos de resposta a estímulos internos e externos, dentre outros. A dinâmica da rede pode ser usada para estudar o processo de divisão e diferenciação celular [Kauffman, 1993]. As redes multiestágio permitem a implementação dinâmica de vários grafos sem a necessidade de resintetizar e reprogramar o FPGA.

Inicialmente, o contexto da aplicação de bioinformática será apresentado na Seção 6.1, onde o modelo do grafo Booleano e o conceito de atratores são apresentados. Posteriormente é introduzido o conceito de grafo livre de escala que é uma topologia mais adequada ao contexto das redes reguladoras [Aldana, 2003]. Em seguida, o algoritmo para cálculo dos atratores é detalhado, juntamente com uma versão sequencial e a análise de complexidade do algoritmo. Finalmente na Subseção 6.1.4, o estado da arte das soluções apresentadas por outros autores.

Após a introdução dos conceitos de atratores nas redes reguladoras, as implementações de redes reguladoras em FPGA propostas por outros autores são apresentados na Seção 6.2. As principais desvantagens das abordagens encontradas na literatura são a natureza estática das soluções e/ou sua validação com grafos pequenos ou apenas em nível de estimativa sem implementações em circuito.

Nesta dissertação, uma nova arquitetura dinâmica é proposta para implementação. A execução de uma versão paralela é apresentada para o cálculo dos atratores em FPGA. Inicialmente o algoritmo é apresentado na Seção 6.3. O algoritmo é implementado em uma arquitetura que é sintetizada uma única vez para grafos de tamanho N. No estudo de redes reguladoras, diversas redes podem ser geradas variando a topologia, as funções e os estados iniciais. Estes pontos são abordados na Seção 6.4.

Na Seção 6.5, os detalhes internos da arquitetura com a implementação das arestas com partições pela rede multiestágios, o controle do algoritmo de cálculo dos atratores e a sua interface com um FPGA Virtex6 VC6VLX240T através do processador softcore Microblaze são apresentados. Um ponto importante a ser destacado que nesta seção as técnicas de roteamento propostas no Capítulo 4 são aplicadas dentro de um contexto de uma aplicação real, onde a configuração das conexões é gerada dinamicamente para cada topologia de grafo que é transmitida para a placa. A interface é um dos gargalos da implementação e pode ser melhorada com trabalhos futuros.

Os resultados experimentais atingiram uma aceleração de 2 a 3 ordens de grandeza da solução proposta em relação à implementação em software sequencial executando em um processador Intel Core2Duo E7400 de 2,8 GHz . Redes de diversos tamanhos foram avaliadas.

6.1 Modelo Grafo Booleano

A capacidade de adaptação dos organismos vivos ao meio ambiente, seu comportamento e sua evolução, são resultados de complexas interações das redes de genes das células. A funcionalidade básica das células é diretamente relacionada à dinâmica destas redes, denominadas redes reguladoras de genes. Nesta dissertação implementamos o modelo de Kauffman [Kauffman, 1969], onde a dinâmica do sistema é determinada pela ativação e desativação dos genes que são os vértices do grafo Booleano. As arestas são responsáveis pela interação entre os vértices.

Formalmente podemos definir uma rede reguladora como um grafo booleano *G(V,A),* onde cada vértice $v_i \in V$ pode ter dois estados (e_i = 0 ou 1). Uma aresta $a_{i->j} \in A$ indica que o vértice v_i atua sobre o vértice v_j. Para cada vértice, uma função booleana f_i calcula seu estado. Suponha que v_i tenha k vizinhos. O estado de v_i no tempo t+1, denominado $e_i(t+1) = f_i(e_{i1}(t),e_{i2}(t),...,e_{ik}(t))$, será calculado em função dos estados dos vértices incidentes no tempo t. Suponha que o grafo tenha N vértices. O estado do grafo é representado pelo conjunto de estados dos vértices $\mathbf{E(t)}=(e_n(t),..,e_i(t),..,e_1(t))$. A Figura 6.1 ilustra uma rede com 3 vértices. A cor preta corresponde ao valor 1 e a cor branca ao valor 0. Suponha uma função de atualização homogênea, definida por um OU-Booleano dos vértices incidentes. Neste exemplo, temos $e_3(t+1)=e_2(t)$, $e_1(t+1)=e_2(t)$, $e_3(t+1)=e_1(t)$ OU $e_3(t)$. Considerando o estado inicial $\mathbf{E}(0)=(001)$ no tempo t=0, teremos a seguinte dinâmica para os estados do grafo, $\mathbf{E}(1)=(010)$ no tempo t=1,

E(2)=(101) no tempo t=2 e no tempo t=3 teremos **E**(3) = (010), ou seja, retorna ao mesmo estado do tempo t=1. A evolução dos estados do grafo é ilustrada na Figura 6.1(a) e o diagrama parcial de estados é mostrado na Figura 6.1(b).

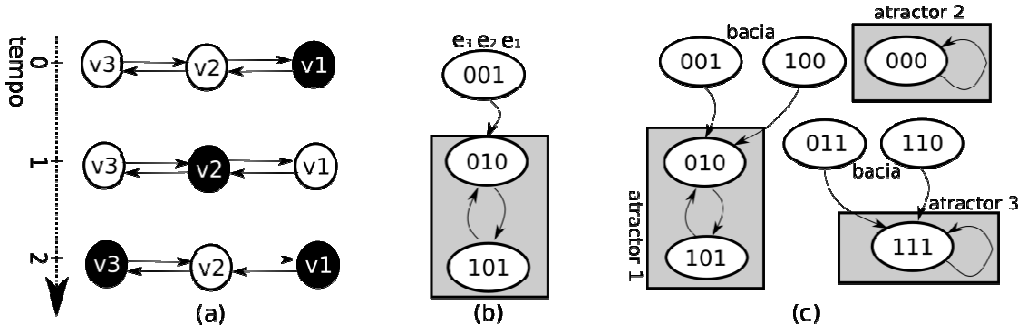


Figura 6.1: Rede Booleana de 3 vértices e seu diagrama de estados: (a) Evolução da rede Booleana no tempo (b) Diagrama parcial de transição de estados (c) Diagrama completo de transição com 3 atratores e suas bacias.

A dinâmica do sistema estuda a evolução dos estados do grafo, os atratores e as suas bacias. Um atrator representa a estabilidade de uma rede regulatória. Os atratores são mais fáceis de serem observados experimentalmente e podem ter uma interpretação funcional associada [Geard, 2009][Kauffman, 1993]. Formalmente, um atrator é um estado ou um conjunto de estados para o qual o sistema tende após um determinado tempo. A Figura 6.1(b) ilustra o diagrama de estados de um atrator, destacado com um retângulo cinza. A bacia de um atrator é um conjunto de estados que irá convergir para um determinado atrator. No exemplo da Figura 6.1(b), o estado 001 pertence à bacia do atrator. Um sistema pode ter vários atratores. As bacias de atratores são os estados transientes iniciais que irão convergir para um determinado estado estável, o atrator. A Figura 6.1(c) ilustra o diagrama completo de todas as transições possíveis para a rede Booleana da Figura 6.1(a). Podemos observar três atratores destacados. O atrator *1* tem 2 estados ou período igual a 2 e os atratores *2* e *3* tem apenas 1 estado. Os atratores *1* e *3* tem uma bacia com 2 estados e o atrator *2* não tem bacia. Neste capítulo é proposto uma implementação paralela para encontrar os atratores e calcular o seu período.

6.1.2 Grafo Livre de Escala

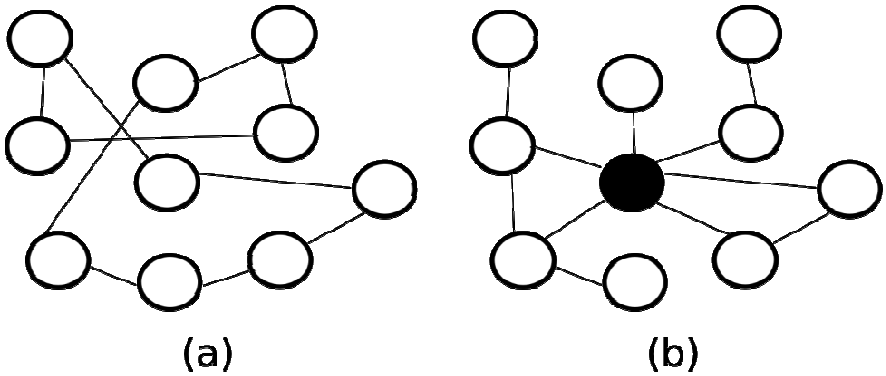


Figura 6.2: (a) Rede Aleatória com K=2 em todos os vértices; (b) Rede Livre de Escala

O modelo de rede proposto em [Kauffman, 1969] e [Kauffman, 1993] faz algumas simplificações: (1) todo gene é conectado à K vizinhos; (2) os K vizinhos são escolhidos com uma distribuição aleatória uniforme. A Figura 6.2(a) ilustra uma rede aleatória onde cada vértice tem dois vizinhos. Entretanto na natureza a topologia de rede livre de escala é muito mais frequente que a aleatória, como demonstram os estudos de redes como Internet, as redes de citações de artigos científicos, as redes sociais, dentre outras [Barabásie, 1999][Watts e Strogatz, 1998]. Um estudo das propriedades das redes reguladoras com topologias livre de escala foi apresentado em [Aldana, 2003]. Primeiro temos que em muitos organismos alguns poucos genes são conectados a um número grandes de genes, e a maioria dos genes possui poucas conexões. A Figura 6.2(b) ilustra um grafo livre de escala com um vértice central destacado que possui muitas ligações, comumente denominado por hub. Além disso, a topologia livre de escala é mais robusta que a redes aleatórias que necessitam de um ajuste fino para encontrar a estabilidade.

6.1.3 Algoritmo para cálculo do período dos atratores

Um algoritmo para cálculo do período de um atrator, sem armazenar a sequencia, foi apresentado em [Bhattacharjya, 1996], baseado na solução de um exercício com sequencias de números apresentado em [Knuth, 1969]. Suponha o modelo síncrono, onde todos os vértices do grafo atualizam simultaneamente seu estado no tempo t+1 em função dos valores dos vértices vizinhos no tempo t. No modelo síncrono, cada estado terá um único sucessor e cada estado aparece apenas uma vez dentro da sequencia do atrator. Baseado nesta

propriedade, o algoritmo começa com duas instâncias da rede, S_0 e S_1, no mesmo estado. A cada passo, a instância S_0 avança uma unidade de tempo e a instância S_1 avança duas unidades de tempo, como ilustrado na Figura 6.3 para um diagrama com 5 estados onde *a* é o estado inicial. No primeiro passo, calcula-se $S_0(1)=b$ e $S_1(2)=c$. Como S_1 avança com o dobro da velocidade, irá percorrer pelo menos uma vez o ciclo antes de encontrar com S_0. Seja t_0, o ponto onde $S_0(t_0)=S_1(2t_0)$, que ocorre no tempo 3 para o exemplo da Figura 6.3. Na segunda fase do algoritmo, para calcular o período P, deixa-se S_1 estacionário, e avançamos com S_0 com a velocidade 1. Assim quando S_0 e S_1 se encontrarem novamente, onde $S_0(t_0+P)=S_1(t_0)$, a simulação termina e o número de passos percorrido entre o ponto t_0 e t_0+P, determina o período P. No exemplo da Figura 6.3, o período será 3 que corresponde aos estados *c, d* e *e*. Este algoritmo pode ser estendido para calcular também o transiente, que no exemplo da Figura 6.3, tem comprimento 2, que são os estados *a* e *b*.

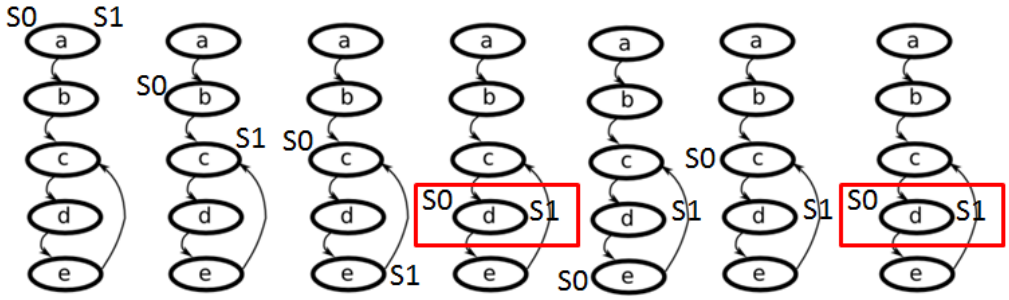


Figura 6.3: Sequência de passos para cálculo do período com duas simulações S0 e S1 para um diagrama de estados parcial.

Para cada passo de simulação, todos os vértices devem ser visitados. Para calcular o novo estado de cada vértice, todos os seus vizinhos devem ser visitados e a função Booleana de atualização do vértice deve ser avaliada. Portanto a cada passo, todas as arestas e vértices serão visitados, ou seja, *O(|V|+|A|)* onde *V* é o conjunto de vértices e *A* o conjunto de arestas. A Figura 6.4 apresenta o pseudocódigo do algoritmo. A função *Passo* percorre a lista de vértices e para cada vértice, a lista de vizinhos é percorrida, ou seja, todos os vértices e arestas são visitados. Para calcular o período, a função *Passo* é chamada várias vezes como ilustra o pseudocódigo *Período* da Figura 6.4. Como o período e o transiente são percorridos pelo menos uma vez temos a complexidade *O((P+T)*(|V|+|A|))*, onde *T* é o transiente.

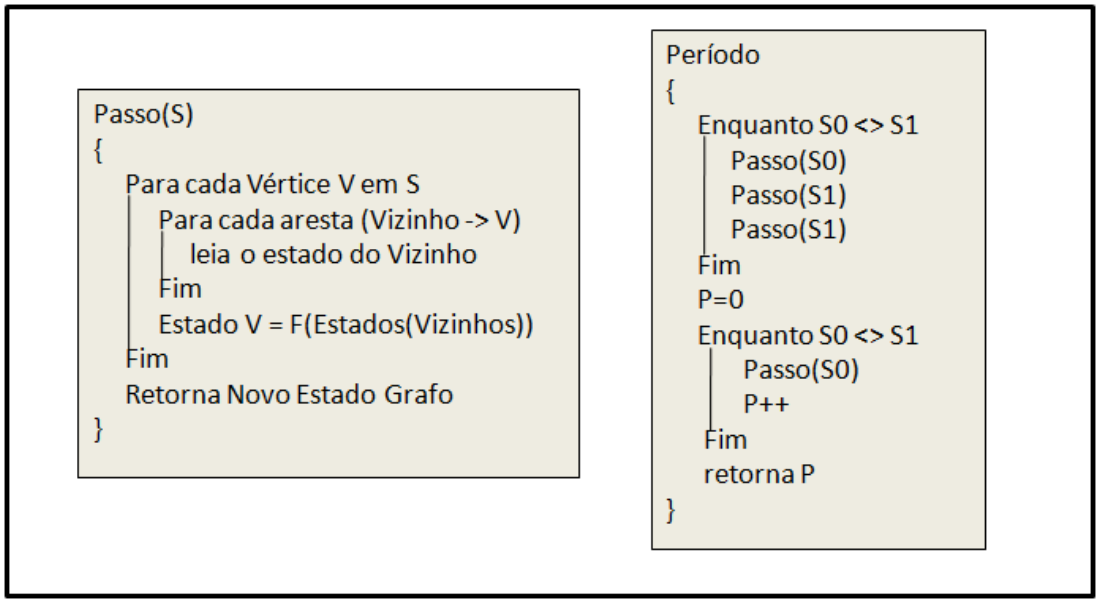


Figura 6.4: Pseudocódigo para cálculo do Período – Versão Sequencial

6.1.4 Trabalhos Correlatos

No trabalho proposto em [Bhattacharjya, 1996], o tempo de execução das simulações não é apresentado, mas redes com altos valores de N, variando de 256 a 50.000 são avaliadas. Porém as redes são restritas a vértices com 2 vizinhos, ou seja k=2, simplificando bem o cálculo do estado do vértice e gerando períodos e transientes pequenos. Além disso, a simulação é abortada após um determinado número de passos, sendo que para valores grandes de N, por exemplo, N=10.000, 20% das amostras de redes não são computadas, por excederam o número máximo de passos. Topologias livres de escala não são abordadas em [Bhattacharjya, 1996]. Como já mencionado, Aldana introduziu as topologias livres de escala na análise das redes reguladoras, porém em seu trabalho [Aldana, 2003] que também não apresenta tempos de execução da simulação e restringe o valor máximo de N a 20 vértices em suas simulações, pois simula todo o espaço de solução. Para N=20 tem-se um milhão de estados.

Outras abordagens baseadas em memória e poda do espaço de solução foram apresentadas em [Garg, 2008] e [Irons, 2007]. Uma simplificação no espaço de busca foi proposta por [Irons, 2006], apresentando um algoritmo com complexidade $O(V^3)$ como caso médio. Redes livres de escala assim como redes aleatórias são avaliadas e resultados com valores de N na ordem de 80 vértices são apresentados, onde funções canalizadoras são usadas nos vértices. O tempo de

execução é da ordem de segundos, sendo de 20 segundos para uma rede com 80 vértices. Porém, este tempo é a mediana do tempo de simulação, se considerar o tempo médio, pois algumas simulações são muito demoradas, o tempo de execução sobe para ordem de 100 segundos. Algumas simulações são abortadas por falta de memória (máximo de 2 milhões de estados armazenados) ou excederem 20 minutos de execução.

O problema de encontrar o período ou os atratores com ciclos curtos em redes booleanas é NP-díficil [Zhang, 2007]. Vários algoritmos para encontrar atratores com período de um único estado são propostos em [Zhang, 2007]. A complexidade para K=2 é $O(1.19^n)$, onde são realizadas podas no espaço de soluções de tamanho $O(2^n)$. O tempo de execução é da ordem de 0,1 segundos para N=80, considerando redes aleatórias e K=2. Para redes livre de escala, a implementação tem tempos menores, da ordem de 1 milissegundo para achar atratores com período 1. Entretanto para atratores com período maior, mesmo com períodos pequenos variando de 2 à 5 estados, a complexidade passa de $O(1.19^n)$ para $O(1.90^n)$ à $O(1.99^n)$. Em 2008 uma solução baseada em diagramas de decisão binária (BDD) para encontrar todo o conjunto de atratores em uma rede Booleana foi proposto por [Garg e Di Cara, 2008]. Os BDDs são eficientes para representar e manipular um conjunto de funções Booleanas. Todas as funções Booleanas da rede são mapeadas no BDD, que pode avaliar rapidamente as transições de estados. Quatro redes pequenas com 10 à 100 vértices, que são redes derivadas das células *mammalian, Th, T-cell*. Além dessas, uma rede sintética com 1263 vértices também é analisada. O tempo de execução para o modelo síncrono é 0.1 à 3 segundos para as redes pequenas e de 200 segundos para a rede grande que tem 5031 arestas, ou seja, com em média 5 vizinhos ou K=5.

6.2 Arquiteturas utilizando FPGA

Vários trabalhos para aceleradores de Bioinformática com FPGA foram propostos. A seguir iremos detalhar os trabalhos mais específicos para manipulação de redes reguladoras e grafos.

Um modelo para representação de grafos em FPGA foi proposto em [Huelsbergen, 2000], onde os algoritmos básicos de grafos como menor caminho, fecho, componentes conexos e detecção de ciclos são implementados. Os grafos podem ser modificados dinamicamente, isto é, inclusão de vértices e aresta sem a necessidade de re-sintetizar. Grafos da ordem de 100 a 1000 vértices foram avaliados e mostraram um potencial de aceleração de três ordens de

grandeza em comparação a solução em um uni-processador. Entretanto, não é possível o cálculo de funções de estados nos vértices, apenas caminhamento pelo grafo. Um projeto de um novo FPGA específico para rede reguladores de proteínas foi apresentado em [Tagkopoulos, 2003]. O novo FPGA combina circuitos analógicos e digitais. A parte analógica usa capacitores para modelar a concentração das proteínas. A parte digital faz a ativação e desativação dos genes. Porém apenas 20 genes podem ser tratados e o novo FPGA precisa ser fabricado, que tem um alto custo. Além disso, apesar de reconfigurável, o novo FPGA é específico para rede de proteínas. Um algoritmo, implementado em um FPGA, baseado em aprendizado bayesiano para rede reguladoras foi proposto em [Pournara, 2005]. Entretanto apenas soluções para redes com 5 e 10 vértices são apresentadas devido a complexidade do método de aprendizado bayesiano. O custo em hardware da solução com 10 vértices é 10 vezes maior que o custo da solução com 5 vértices, evidenciando problemas de escalabilidade da solução. Um algoritmo para cálculo de período em redes booleanas probabilísticas implementado em um processador ARM com o auxílio de um FPGA foi proposto em [Zerarka, 2004]. O tempo de execução é apresentado apenas para um exemplo com três variáveis, da ordem de 400ms. O cálculo dos estados da rede e a detecção de ciclos são realizados no FPGA, onde o grafo booleano é mapeado. Todos os vértices são atualizados em paralelo e em apenas um ciclo de relógio, o próximo estado da rede é calculado, ou seja, complexidade O(1), ao passo que a solução sequencial em um processador tem complexidade O(|V|+|A|). Uma fila armazena os últimos estados e tem o tamanho máximo M. A cada passo, todos os valores da fila são comparados em paralelo para verificar se existe alguma repetição que significa a ocorrência de um atrator. Apenas pequenos atratores são detectados, cujo período é menor que M. Entretanto, para cada instância de rede booleana, o código VHDL do grafo deve gerado e sintetizado no FPGA, cujo tempo de síntese é bem superior ao tempo de simulação. O tempo de síntese não é apresentado. Apesar de apresentar resultados de área ocupado no FPGA para redes com até 800 vértices, apenas uma rede de 3 vértices foi simulada. Além disso, cada vértice do grafo pode ter até 5 vizinhos e topologias livres de escala não são tratadas.

6.3 A arquitetura proposta e Algoritmo Paralelo em Hardware

A solução proposta aqui para FPGA é diferente das anteriores em vários aspectos. Primeiro, não implica no desenvolvimento de um novo circuito de FPGA [Tagkopoulos, 2003], pode tratar um grande conjunto de vértices, não sendo limitada a poucos vértices como

[Tagkopoulos, 2003][Pournara, 2005]. Segundo, apresentamos uma solução paralela para o problema de cálculo do período de atratores de redes Booleanas, semelhante ao trabalho proposto por [Zerarka, 2004]. Entretanto, abordamos redes Booleanas com topologias livre de escala enquanto o trabalho de [Zerarka, 2004] aborda redes probabilísticas. Além disso, o trabalho de [Zerarka, 2004] fica limitado a 5 conexões por vértices, enquanto nossa solução possui flexibilidade para permitir vértices com diferentes configurações. O ponto mais importante é o tempo de síntese, posicionamento e roteamento, que nos trabalhos anteriores não é discutido. Porém é fundamental, pois é em geral ordens de grandeza maior que o tempo de execução dos cálculos [Huelsbergen, 2000]. O FPGA implementa o cálculo de forma rápida e paralela, porém o tempo de síntese é cada vez maior devido a complexidade do FPGA. Nossa solução propõe uma rede de interconexão flexível que permite que vários grafos Booleanos sejam simulados sem a necessidade de resintetizar o circuito, ou seja, temos um nível de reconfiguração rápida para modificar a topologia do grafo. A inserção e remoção de arestas de grafos dinamicamente foram abordadas por [Huelsbergen, 2000], porém os vértices não tem capacidade de processamento.

Este trabalho propõe uma implementação paralela para executar no FPGA. A Figura 6.5 ilustra o pseudocódigo do algoritmo. Semelhante a versão sequencial, o algoritmo trabalha em duas etapas. A função *Vértice* é implementada em cada vértice do grafo e executada em paralelo em todos eles. A função trabalha em dois modos de simulação. No modo *S0* é executada a mudança de estado para a simulação S0 e no modo *S1* é executada a mudança de estado para simulação S1. A função principal para calcular o período executa em uma unidade controladora separada dos vértices. Ao executar o *modo S0* ou o *S1*, cada vértice envia o resultado da comparação de S0 e S1 para a unidade controladora, que recebe os sinais de todos os vértices, e verifica se S0 é igual a S1. A unidade controladora recebe os sinais e contabiliza o período. Na versão atual foi implementada a função majoritária, onde o novo valor do vértice é 0, se a maioria dos vizinhos tem o valor 0, caso contrário será 1. Cada aresta incidente pode inverter ou não o sinal do vértice vizinho. Estudos recentes mostram que a função majoritária é vantajosa para ajuste das propriedades das redes reguladoras [Darabos, 2011].

```
Vertice(Modo)
{
  Se (Modo == S0)
    Envia valor do estado S0
    Recebe sinais dos vizinhos
    Calcula novo valor do estado S0
    Grava novo valor do estado S0
  Se (Modo == S1)
    Envia valor do estado S1
    Recebe sinais dos vizinhos
    Calcula novo valor do estado S1
    Grava novo valor do estado S1
}
```

```
Periodo
{
   Enquanto S0 <> S1
     Para todos os vértices em paralelo
       Passo(S0)
       Passo(S1)
       Passo(S1)
   Fim
   P=0
   Enquanto S0 <> S1
      Para todos os vértices em paralelo
        Passo(S0)
      P++
   Fim
   retorna P
}
```

Figura 6.5: Pseudo-Código para cálculo do Período – Versão Paralela para FPGA

A vantagem da implementação em hardware pode ser vista na Figura 6.6, que ilustra um modelo simplificado do vértice. Cada vértice tem um flip-flop para armazenar o valor do estado atual e uma função de atualização. A cada ciclo de relógio, o vértice recebe os estados dos vértices vizinhos e atualiza seu novo estado, ou seja, em um único ciclo de relógio é possível calcular o novo estado para todos os vértices do grafo. Em contraste com a versão em software em um único processador onde é necessário visitar todas as arestas, gastando vários ciclos de relógio e vários acessos à memória. Além do modelo apresentado na Figura 6.6, cada vértice possui uma máquina de estados para implementar o pseudocódigo da Figura 6.5. São necessários dois flip-flop, um para a instância S0 e outro para a instância S1, mas apenas uma função de atualização que pode ser compartilhada por S0 e S1.

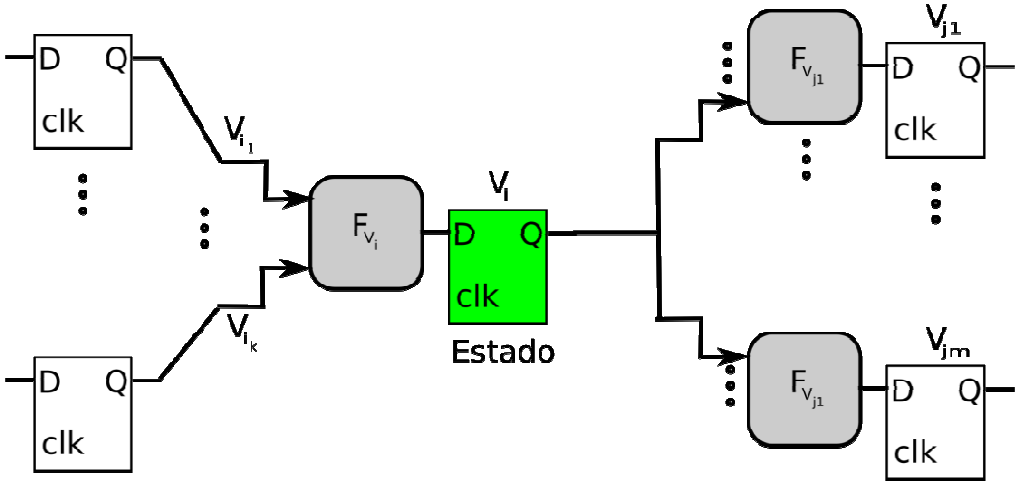


Figura 6.6: Modelo do Vértice em hardware

6.4 A síntese em FPGA

Uma alternativa de implementação em FPGA para o algoritmo da Figura 6.5, é a síntese direta mapeando vértices e conexões em hardware. Esta abordagem é utilizada por trabalhos anteriores [Pournara, 2005][Zerarka, 2004]. A unidade controladora e o conjunto de vértices podem ser sintetizados juntamente com o conjunto de interligações que implementam as arestas, e a comunicação com a unidade controladora. Entretanto, o tempo de síntese é elevado e em função do tamanho do grafo, pode ser proibitivo. A Figura 6.7 ilustra os passos. Primeiro, o grafo é gerado, com uma distribuição livre de escala para as arestas. Os vértices, arestas e unidade controladora são sintetizados. Depois, o FPGA é configurado e finalmente, dado um conjunto inicial de estados para os vértices, o algoritmo é executado. O tempo de síntese será o passo mais lento do processo. Além disso, para simular um novo grafo, todo o processo deve ser repetido. A única vantagem dessa alternativa é no caso de estudar o comportamento da rede para diferentes estados iniciais, sem modificar a função de atualização dos vértices nem o conjunto de arestas. Neste caso, basta inicializar os vértices com um novo valor e executar a simulação.

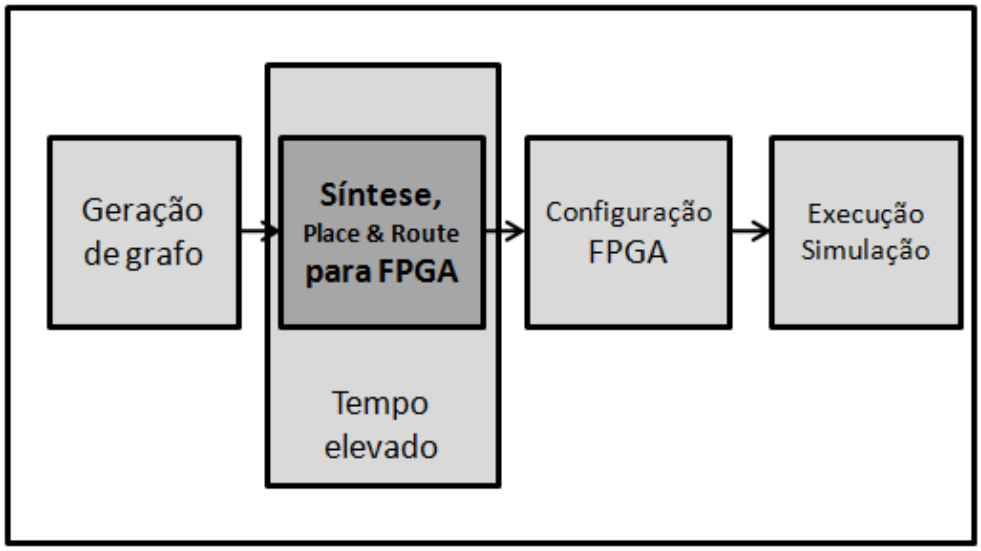


Figura 6.7: Diagrama para implementação dos vértices e conexões no FPGA

Porém, no estudo de redes reguladoras é necessário gerar vários grafos diferentes para obter uma amostragem significativa das características das redes, como número de atratores, tamanho médio e máximo do transiente, período, etc. O que não é viável na solução proposta na Figura 6.7, onde o grafo é fixo no FPGA. Como já comentamos, neste trabalho é proposta uma abordagem diferente, onde a síntese no FPGA é realizada uma única vez, e para gerar um novo grafo basta re-configurar dinamicamente as conexões que implementam as arestas.

A solução proposta por este trabalho é a síntese de uma arquitetura genérica para um grafo com N vértices. A síntese é realizada uma única vez, como ilustrado na Figura 6.8. Um conjunto de vértices com diferentes tamanhos é gerado. As arestas são implementadas por uma rede de interconexão dinâmica, implementada por uma rede multiestágio Omega radix4.

Uma segunda etapa, ilustrada na Figura 6.9, gera vários grafos diferentes, para cada grafo, a palavra de configuração da rede de interconexões é gerada, enviada ao FPGA e a simulação pode ser executada para diferentes estados iniciais. Ou seja, não é necessário re-sintetizar novo grafo, basta modificar a configuração da rede de interconexões. Esta solução introduz um nível de configuração dinâmica acima do FPGA.

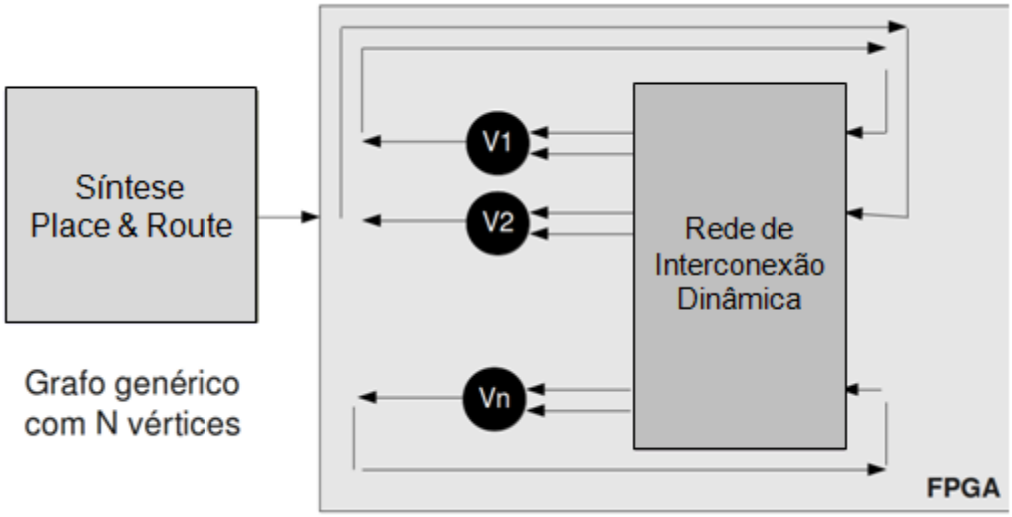


Figura 6.8: Arquitetura Genérica para um grafo com N vértices e um conjunto de arestas reconfigurável

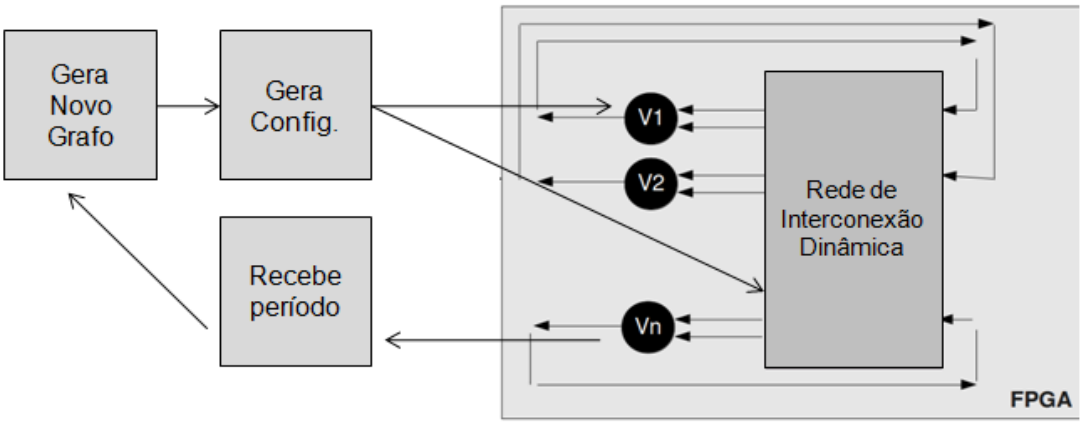


Figura 6.9: Geração de vários grafos sem a necessidade de re-sintetizar

6.5 Arquitetura e interface com a Placa Virtex6

A arquitetura proposta foi desenvolvida em VHDL e implementada em uma Virtex6. Onde mostrou-se completamente funcional. Para facilitar a comunicação da arquitetura com o computador, utilizou-se o soft-processor da Xilinx, o Microblaze para comunicação, envio do grafo e recebimento do período e transiente. A Figura 6.10 ilustra o conjunto completo junto com a arquitetura proposta que é executada dentro do FPGA.

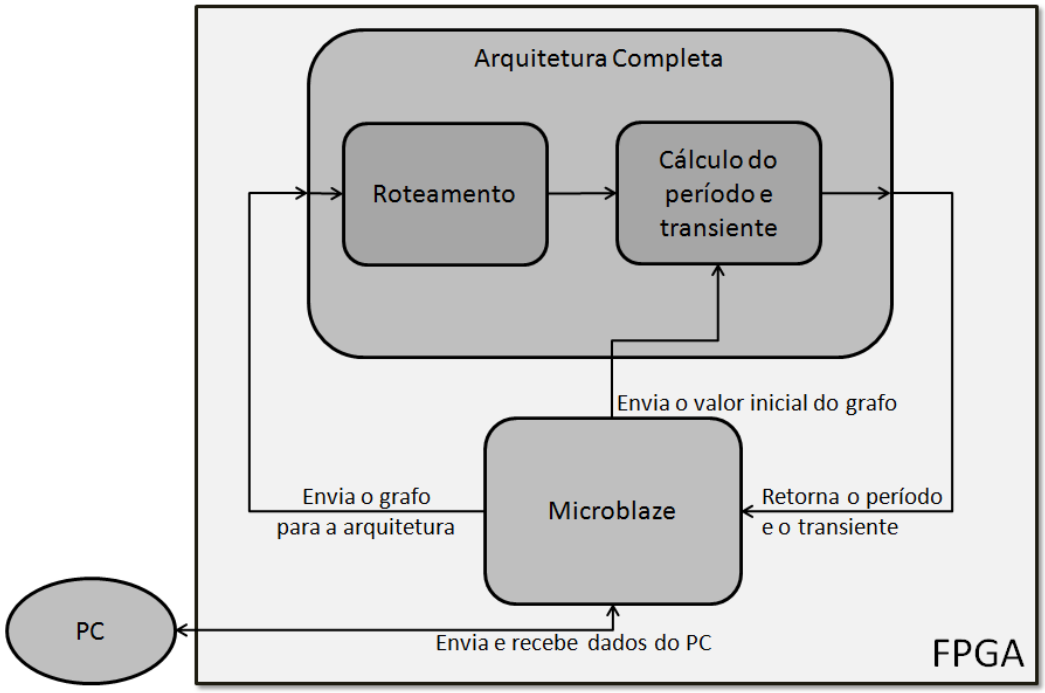


Figura 6.10: Arquitetura implementada utilizando o Microblaze para comunicação com o computador.

Como podemos observar na Figura 6.10, o papel do Microblaze é apenas comunicar com o computador e com a arquitetura proposta. A comunicação entre Microblaze e computador é feita através de uma comunicação serial, RS-232 de 921Kbps. Como esta não é uma comunicação muito eficiente, só é enviado e recebido o mais essencial possível na execução do algoritmo. Já a conexão entre a arquitetura e o Microblaze é feita utilizando o barramento PLB (*Processor Local Bus*), que é acessado através de endereços de memória do Microblaze. Podemos ver também que o Microblaze é responsável por enviar o valor inicial do grafo, que pode ser vários estados iniciais gerados aleatoriamente. Enviando um estado inicial, a arquitetura retorna o período e o transiente, após isso o Microblaze envia outro estado inicial. O Microblaze também é responsável por enviar o grafo para a arquitetura poder rotear o mesmo para ser executado o algoritmo utilizando a rede multiestágio. Toda a arquitetura no

FPGA é executada a uma frequência de 90 Mhz. É possível aumentar este valor com algumas técnicas de otimização e *pipeline*, que por questões de complexidade e tempo não foram utilizadas neste trabalho.

A Figura 6.11 nos dá um maior detalhe sobre a arquitetura proposta:

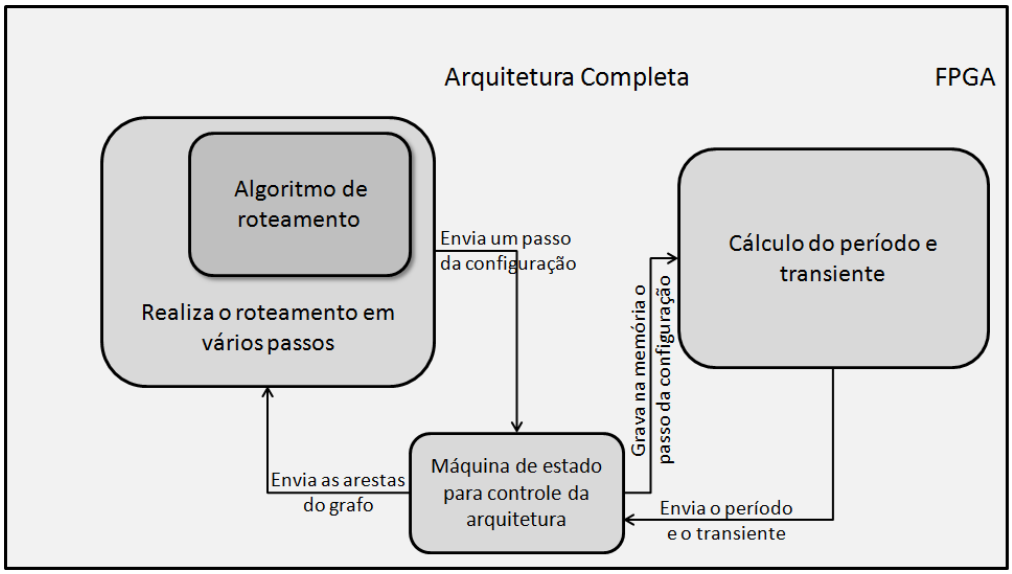


Figura 6.11: Arquitetura Completa com mais detalhes

Aqui podemos ver que junto com a arquitetura de cálculo do período temos a arquitetura que faz o roteamento da rede multiestágio, utilizando apenas uma rede e *multicast*, a arquitetura de roteamento já foi abordada no Capítulo 4 e não será muito comentada aqui. Além das arquiteturas, temos uma máquina de estado responsável por todo o controle de enviar o grafo para ser roteado e se adequar a arquitetura de cálculo do período. Ela também é responsável por enviar as configurações da rede depois de pronta para a rede multiestágio. A máquina de estado também lê o período e o transiente e envia para o Microblaze. Lembrando que toda a arquitetura descrita na Figura 6.11 é executada no FPGA.

Na arquitetura de cálculo do período existe uma rede multiestágio e vários vértices que fazem o cálculo dos seus estados. A Figura 6.12 detalha isso:

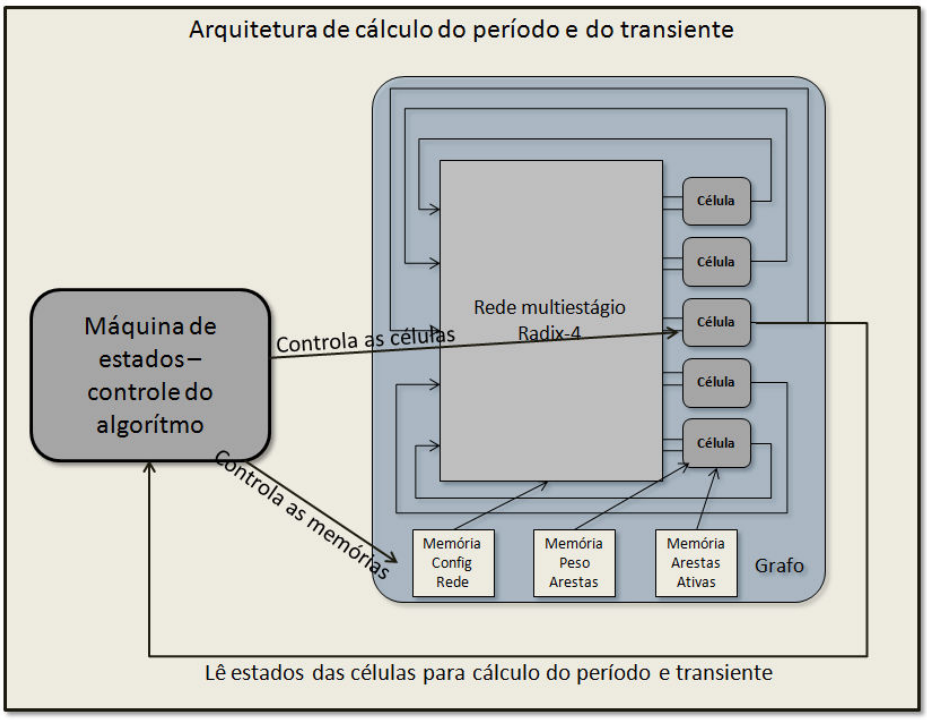


Figura 6.12: Arquitetura de cálculo do período e transiente

Podemos ver na Figura 6.12, que a arquitetura possui uma rede radix4, e várias células ligadas nela. Além disso, temos 3 memórias de configuração, uma para configurar o roteamento da rede, uma que possui o peso das ligações(arestas) do grafo, e uma memória para informar qual ligação da célula está ativa naquele determinado momento, isso é importante quando temos grafos que não utilizam todas as arestas. Temos também a máquina de estados que faz todo o controle de qual configuração colocar em cada instante e de verificar se os estados S0 e S1 são iguais. É importante ressaltar que esta arquitetura pode dividir o conjunto de arestas do grafo em várias partições, isso se dá para não precisarmos criar uma rede muito grande que ocupe muitos recursos do FPGA. Assim um grafo com 500 arestas e 256 vértices poderia ser representado por uma arquitetura com 256 vértices e rede de tamanho 256, pois a arquitetura criaria mais de uma configuração ou partição nas memórias e essas seriam programadas de forma adequada. Se não fosse assim, teríamos que ter uma rede de pelo menos 512 entradas e saídas, e não termos praticamente conflito nenhum nas conexões para termos todas as arestas roteadas, para isso seria necessário estágios extras o que também aumentaria o custo da rede.

Com essa capacidade de particionamento, a Figura 6.13 ilustra como a parte de roteamento foi estruturada:

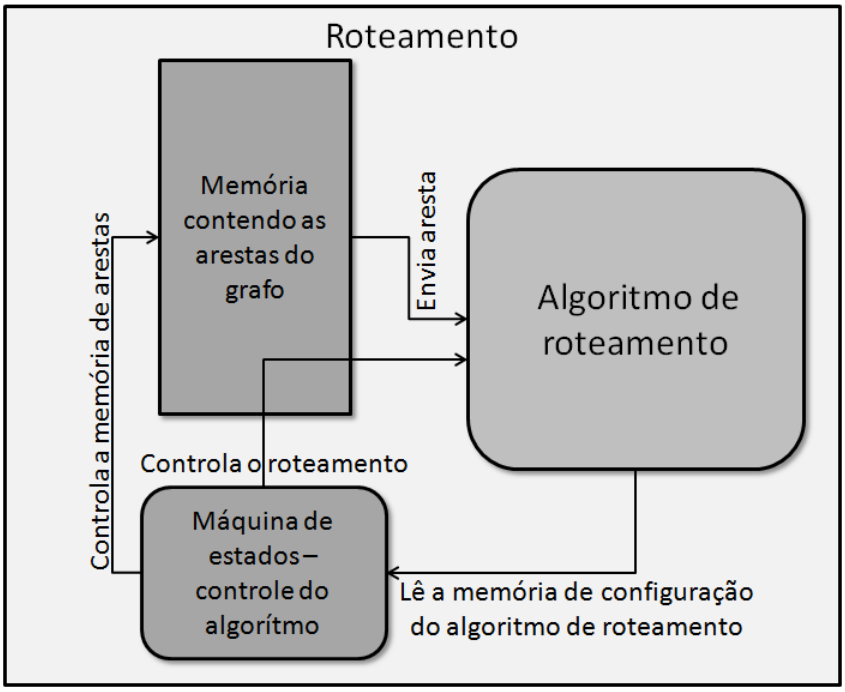


Figura 6.13: Arquitetura responsável por criar as configurações das memórias para o cálculo do período.

Podemos ver na Figura 6.13, que além do roteamento propriamente dito, temos uma memória onde fica armazenado o grafo que será calculado, e também temos uma máquina de estado que controla o roteamento e é responsável por particionar o grafo em configurações e as enviar para a arquitetura de cálculo do período.

O roteamento foi implementado em hardware porque o tempo de execução é crítico e todos os gargalos devem ser evitados, assim como o algoritmo do cálculo do período roda muito rápido, se o roteamento fosse feito utilizando o Microblaze, o ganho de desempenho geral da arquitetura ia ficar muito prejudicado, diminuindo muito. Com isso, foi necessário desenvolver o roteamento em hardware para não termos o roteamento como gargalo deixando a arquitetura a mais eficiente possível.

6.6 Resultados Experimentais

Para desenvolver esse trabalho, foi utilizado a ferramenta EDK 12.4 da Xilinx para integrar a arquitetura no Microblaze, além da ISE 12.4 para programação dos códigos em VHDL das arquiteturas implementadas. Para desenvolvimento dos códigos em C++ do Microblaze foi utilizado o SDK 12.4 da Xilinx, que tem como base o Eclipse, e para códigos C++ para executar

em computador foi utilizado o DEV-C++ com o compilador MINGW. Todas as ferramentas aqui descritas executaram em ambiente Microsoft Windows.

Os testes no FPGA foram executados em uma placa de prototipação e desenvolvimento da Xilinx, a ML605, com um chip FPGA Virtex-6 XC6VLX240T-1FFG1156, mais informações em [ML605 Site]. Os testes utilizando o algoritmo sequencial foram executados em um computador com um processador Intel Core2Duo E7400 de 2,8 GHz, com 4 Gigabytes de memória RAM e em ambiente Microsoft Windows XP.

A arquitetura desenvolvida para obtenção dos resultados foi a descrita em alto nível nas Seções 6.5, porém com alguns limites físicos. Estes limites podem ser reduzidos aumentando o tamanho da arquitetura. A arquitetura é capaz de trabalhar com grafos de até 256 vértices e com até 8192 arestas. É capaz de armazenar até 64 partições do grafo. A arquitetura possui uma rede radix4 com 256 entradas, com *multicast* e sem estágios extras, ou seja, temos uma E/S da rede para cada vértice. E como já falado a arquitetura executa a 90 MHz. Todos estes parâmetros podem ser alterados na arquitetura para trabalharmos com grafos maiores, porém não é garantido que a frequência de operação de 90 MHz poderá ser atingida. Essa frequência de operação também foi conseguida com base na arquitetura de FPGA utilizada a Virtex6 XC6VLX240T-1FFG1156, se utilizarmos arquiteturas diferentes, é possível não obter a mesma frequência de operação, ou então atingirmos frequências de operação maiores, tendo então um desempenho diferente do obtido nesse experimento. O mesmo vale para a versão sequencial, onde o desempenho varia de acordo com o processador utilizado.

O Microblaze foi montado com apenas um bloco de memória de 256 Kbytes, e o módulo de comunicação RS-232 para comunicar com o computador, todos os outros componentes disponíveis foram removidos para diminuir o atraso gerado pelos mesmos e obter um ganho na frequência de operação. A Tabela 6.1 possui o tamanho da arquitetura sintetizada:

Tamanho Síntese			
Componente	Flip Flops	LUTs	BRAMs
Arq. Proposta	9842	10980	87
RS-232	142	124	0
Microblaze	2245	2981	1
Bloco de memória	6	8	64

Tabela 6.1: Tamanho dos componentes na síntese da arquitetura.

Para conseguir completar com sucesso o place & route da arquitetura, Foi utilizado a ferramenta SmartXplorer, com a técnica mapextraeffort2. A Tabela 6.2 possui o tamanho da arquitetura antes e depois de ter sido feito o posicionamento e o roteamento da arquitetura:

	Flip Flops	LUTs	BRAMS
Tam. Arq. Sintese	12575	14633	152
Tam. Arq. P&R	12247	13035	152
Tam. FPGA	301440	150720	416
Perc. Usado sintese	4,2%	9,7%	36,5%
Perc. Usado P&R	4,1%	8,6%	36,5%

Tabela 6.2: Custo da arquitetura em uma Virtex-6, apenas sintetizada e com place & route.

Como podemos observar na Tabela 6.2, o place & route acabou deixando a arquitetura um pouco menor, esses resultados de tamanho variam com a técnica utilizada na hora de realizar o place & route. Neste caso tivemos a vantagem de ter uma arquitetura menor que a síntese, o que nem sempre pode ocorrer. Outro importante detalhe que vemos com esses resultados é que a arquitetura é relativamente pequena se comparada ao tamanho total do FPGA. Ocupando menos de 10% das LUTs do FPGA e menos de 5% dos registros do FPGA. O consumo de memória embarcada foi relativamente alto comparando com os outros recursos, porém é possível utilizar menos memórias embarcadas utilizando LUTs como memória. Assim a arquitetura utilizará mais espaço no FPGA, porém ocupará menos blocos de memória. Portanto seria possível executar mais instancias do algoritmo colocando várias arquiteturas executando em paralelo, deixando apenas um Microblaze gerenciar o restante das operações, gerando um ganho proporcional ao número de implementações paralelas dos grafos no desempenho geral do sistema.

A seguir temos o resultado de desempenho da arquitetura executando no FPGA. Foram feitos testes de cálculo de período com grafos variando de tamanho 20 até tamanho 256, e arestas de 40 até 8149 arestas. Os resultados foram comparados com o algoritmo sequencial desenvolvido executados em um processador Intel Core2Duo E7400 de 2,8 GHz pelo aluno Leonardo Fonseca de Carvalho, aluno de mestrado do departamento de Informática da UFV. Nas tabelas, Tabela 6.3, Tabela 6.4 e Tabela 6.5 há informações sobre os grafos testados:

Número de arestas		TAMANHO				
		20	50	100	150	256
GAMA	2.0	40	244	610	801	1798
	1.8	83	320	512	1313	2270
	1.6	77	320	933	2015	3114
	1.4	123	417	1294	2631	4649
	1.2	122	589	1708	2519	8149

Tabela 6.3: Número de arestas dos grafos testados.

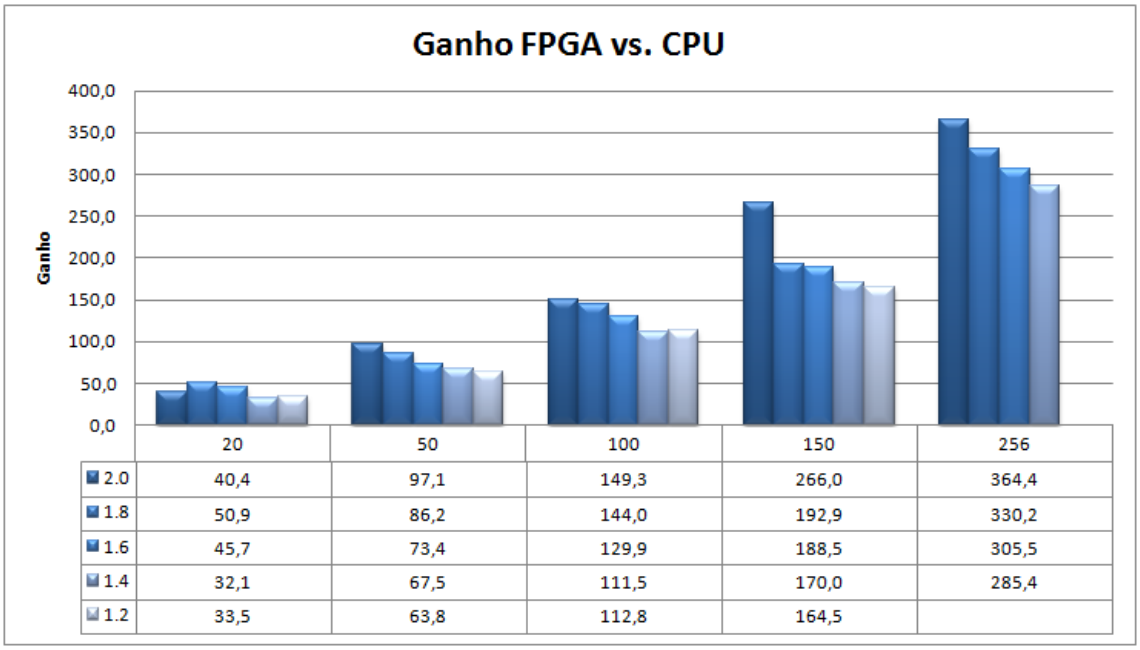
Período		TAMANHO				
		20	50	100	150	256
GAMA	2.0	10	12	36	69	1080
	1.8	9	87	92	972	8184
	1.6	16	22	69	3775	126531
	1.4	14	50	1313	246300	1128518
	1.2	9	29	814	5779	553248383

Tabela 6.4: Tamanho do período dos grafos testados.

Transiente		TAMANHO				
		20	50	100	150	256
GAMA	2.0	11	36	93	801	1001
	1.8	10	68	66	1351	10830
	1.6	16	35	593	2015	48203
	1.4	11	26	2793	66376	1223078
	1.2	10	99	2000	3472	65465100

Tabela 6.5: tamanho do transiente dos grafos testados.

Como podemos observar nas tabelas temos grafos de vários tipos, e com vários tamanhos de períodos e transientes, variando o tamanho do período de 9 até períodos na casa dos 500 milhões, onde são períodos enormes. Os transientes também variam bastante de 10 até a casa dos 60 milhões. É importante ressaltar que esses valores foram obtidos com base em um estado inicial aleatório, é se variássemos o estado inicial, teríamos outro valor de transiente, e poderíamos ter outro valor de período se esse caísse em uma bacia diferente. Esses dados foram obtidos somente para fazer a comparação de ganho de desempenho comparando os dois algoritmos, o executado no FPGA e o executado no Core2Duo. A Figura 6.14 mostra o ganho do FPGA em tempo de execução em relação ao algoritmo sequencial.



	20	50	100	150	256
2.0	40,4	97,1	149,3	266,0	364,4
1.8	50,9	86,2	144,0	192,9	330,2
1.6	45,7	73,4	129,9	188,5	305,5
1.4	32,1	67,5	111,5	170,0	285,4
1.2	33,5	63,8	112,8	164,5	

Figura 6.14: Ganho do algoritmo em hardware contra do em software.

Como podemos ver na Figura 6.14 a arquitetura proposta obteve uma velocidade de execução do algoritmo de até 364 vezes em relação ao Intel Core2Duo E7400 de 2,8 GHz. E no pior caso, 32 vezes. É normal a arquitetura ter um ganho menor com grafos pequenos, pois ela fica subutilizada já que muito dos seus vértices ficam parados, e os vértices ativos ainda continuam com apenas 1 E/S da rede. Apesar do tempo de execução do algoritmo sequencial ser rápido, variando de milissegundos a minutos para um determinado grafo, em situações de uso do sistema, milhares de grafos são avaliados. Na Tabela 6.6 e Tabela 6.7 temos os valores em tempo das execuções dos algoritmos:

execução sequêncial		TAMANHO				
milissegundos		20	50	100	150	256
GAMA	2.0	0,2322	2,175	12,75	28	410
	1.8	0,3856	5,666	11,39	365	5473
	1.6	0,4453	2,637	80,39	717	45000
	1.4	0,3769	3,111	553,35	57000	948000
	1.2	0,4106	7,769	408,91	1539	-

Tabela 6.6: Tempo de execução dos grafos na versão sequencial, em milissegundos.

execução FPGA		TAMANHO				
milissegundos		20	50	100	150	256
GAMA	2.0	0,005744	0,022411	0,085411	0,105244	1,125244
	1.8	0,007578	0,065744	0,079078	1,891744	16,57708
	1.6	0,009744	0,035911	0,618744	3,803411	147,2864
	1.4	0,011744	0,046078	4,963078	335,3869	3321,103
	1.2	0,012256	0,121744	3,625244	9,353244	1198005

Tabela 6.7: Tempo de execução dos grafos no FPGA, em milissegundos.

Como podemos analisar através destes resultados, os grafos pequenos executam bem rápido na versão sequencial do algoritmo, porém é importante lembrar que o algoritmo é executado várias vezes com estados iniciais diferentes, assim qualquer ganho de tempo é bem-vindo. Já para grafos grandes, com muitas arestas e períodos grandes, o tempo de execução é considerável. Chegando a gastar 15 minutos para uma execução, como podemos ver o grafo com Gama 1.4 e tamanho 256. Já no FPGA esse tempo foi de apenas 3,3 segundos. Vale destacar que o grafo com Gama 1.2 e tamanho 256 não se obteve uma resposta na versão sequencial em 24 horas de execução. Tomando por base o tempo do FPGA, que foi de praticamente 20 minutos, se o ganho em aceleração fosse de 200 vezes, o tempo total de execução na versão sequencial seria de mais de 66 horas, o que equivale a 2 dias e 18 horas.

6.6 Considerações Finais

A arquitetura proposta foi implementada em FPGA, e mostrou ser viável com ganhos de até 360 vezes sobre o algoritmo executado no processador Intel Core2Duo E7400 de 2,8 GHz. Assim é possível analisar um espaço de soluções muito maior no mesmo espaço de tempo. A arquitetura ocupou menos de 10% de LUTs do FPGA utilizado, assim é possível rodar mais instâncias da arquitetura em paralelo para ter um ganho maior de desempenho. Trabalhos futuros podem ser realizados com técnicas para otimizar a arquitetura, como o uso de *pipeline*, para alcançar frequências mais altas de operação, melhorias na interface de comunicação com o uso de barramentos PCI ou memória DDR disponível na placa de FPGA utilizada.

7. Conclusões

Este trabalho analisou as redes multiestágio Omega em FPGA para execução de aplicações de alto desempenho com reconfiguração dinâmica utilizando uma camada virtual. Desenvolvemos algoritmos em hardware para o roteamento das redes, e utilizamos a camada virtual sobre FPGA comercias em duas arquiteturas, uma para uma arquitetura de grão-grosso, e outra em bioinformática.

Primeiro verificamos que as redes de interconexão multiestágio possuem um comportamento regular se utilizadas em FPGAs, o que facilita o uso das mesmas na arquitetura, já que podemos prever seu tamanho e sua latência. Ou seja, a partir de uma descrição parametrizada em VHDL da rede (comutadores e suas ligações), a ferramenta de síntese mapeou a implementação em LUT com um custo O(N Lg N) para área e O(Lg N) + S para a latência. Para as gerações atuais de FPGA como a linha Virtex6 com LUT de 6 entradas, mostrou-se que a implementação com radix4 reduz o custo em função da equivalência exata entre um comutador e um conjunto de 4 LUTs.

Para reduzir o custo, optamos por uma rede Omega. Entretanto, a rede Omega é bloqueante. Diferente da rede crossbar que pode realizar qualquer ligação sem conflito, porém tem complexidade de crescimento $O(N^2)$. Para contornar o problema de bloqueio nas ligações, a maioria dos trabalhos na literatura utiliza a rede Omega para alguns padrões de comunicação que não geram bloqueios. Nesta dissertação mostramos que se a rede for utilizada com permutações parciais, isto é,com ma menor carga de trabalho, o seu comportamento se torna não bloqueante. Com essa possibilidade de utilização das redes, fez-se um estudo detalhado sobre sua capacidade de roteamento. Capacidade esta, próxima à redes rearranjáveis. Ademais, diferente das redes rearranjáveis que são usadas em um contexto onde se conhece a priori o conjunto de ligações, nesta dissertação avaliou-se as redes Omega e mostrou-se que elas podem ser usadas sob demanda (on-the-fly) em ambientes dinâmicos. Além disso, utilizando duas redes em paralelo com estágios extras, tem-se praticamente o mesmo custo de uma rede rearranjável, uma latência menor e um comportamento não bloqueante. Outro ponto verificado no estudo da capacidade de roteamento foi seu uso em um contexto com *multicast*, e isto não afeta a capacidade de roteamento da rede.

Um dos temas mais estudados sobre as redes multiestágios nas décadas de 80/90 foram algoritmos paralelos para configurá-las no contexto de roteamento por circuito. Nesta dissertação não foi abordado o roteamento por pacotes. A motivação principal do uso das redes é sua latência O (Lg N) em projetos de máquinas paralelas SIMD e MIMD, onde as redes eram usadas para ligar os processadores aos bancos de memória ou para interconectar os processadores. Porém para configurar as redes, o tempo de execução dos algoritmos era no mínimo O(N log N). Além disso, na época os trabalhos tiveram um cunho teórico, sem implementações com algoritmos e analise de complexidade em alto nível pela não disponibilidade de máquinas paralelas. Mais ainda, o algoritmo precisa muitas vezes executar em um computador paralelo que não era especificado com detalhe nos trabalhos. Com a utilização de hardware programável, no nosso caso FPGA, é possível avaliar as implementações no nível físico. Desenvolvemos dois algoritmos de roteamento em hardware, para aplicações onde o roteamento da rede onde o tempo de execução é crítico (da ordem de poucos milissegundos) e há necessidade de roteamento *Just-in-time*.

O potencial do uso das redes foi a motivação para buscar aplicações em arquiteturas paralelas. Vimos que essas redes são funcionais para realizar interconexões e criar camadas virtuais de reconfiguração para arquiteturas de grão grosso, dando continuidade a trabalhos anteriores [Ferreira, 2011]. Além disso, mostrou-se que elas podem ser implementadas de forma eficiente em FPGA. Com os algoritmos de roteamento em hardware, mostrou-se que é viável calcular a reconfiguração da arquitetura em tempo de execução. Com isso seu uso pôde ser integrado em um algoritmo de escalonamento, posicionamento e roteamento para aplicações de multimídia com alto grau de paralelismo descritas com grafos de fluxo de dados.

O uso em problemas de bioinformática que demandam uma grande quantidade de cálculo também pode ser vantajoso. Especificamente, para o problema de cálculo de atratores em redes reguladoras de genes, mostrou-se que a solução baseada em uma arquitetura paralela com a rede multiestágio pode gerar um ganho de aproximadamente 360 vezes comparado com um Core2Duo executando código sequencial.

Pode-se concluir então que as redes multiestágio são uma solução eficiente para utilização em arquiteturas paralelas, onde há necessidade de roteamento rápido e sob demanda. Possuem tamanho e latência menor que redes *crossbar*, complexidade de roteamento baixa O(Lg N) e podem utilizadas para criar camadas virtuais para gerar arquiteturas paralelas reconfiguráveis,

gerando ganhos de desempenho de duas ou três ordens de grandeza, como mostramos nos Capítulos 5 e 6.

Como trabalho futuro propõe-se melhorar a arquitetura de bioinformática, para esta alcançar melhores freqüências de operação, usando técnicas de otimização e *pipeline*. O modo de ligação entre a rede e suas memórias de configuração é direta, o que pode deixar a comunicação com os comutadores com uma alta latência. O que limita o uso de redes maiores que as testadas. É necessário um estudo de novos métodos de armazenamento das configurações sem ser em memória global para evitar essa alta latência.

Outra questão é a geração de números aleatórios que é utilizada para criar os estados iniciais dos grafos. Atualmente ela é feita no Microblaze, o que dependendo do tamanho do período do grafo, pode ser um gargalo para o sistema. Uma sugestão seria o desenvolvimento de uma arquitetura em hardware para geração de números aleatórios.

O tempo de carregamento do grafo para o FPGA pode ser melhorado utilizando outras técnicas de comunicação, como o barramento *PCI Express*, além disso, é importante ressaltar que o ganho da arquitetura de roteamento foi pouco expressivo, pois, nela havia um overhead da cópia das arestas e após cada roteamento, a configuração da rede que era criada se transmitia para as memórias de configuração da rede. E todo este tempo era contabilizado no tempo do roteamento. Para interligar o Microblaze e a arquitetura desenvolvida, foi utilizado o barramento PLB (*Processor Local Bus*), para cada informação de 32 bits enviada ou recebida da arquitetura, o sistema gastava 10 ciclos de relógio. A substituição do barramento de interligação da arquitetura por outro mais eficiente pode evitar esse overhead.

Outra alternativa para melhorar o tempo de execução é otimizar o *Place & Route* da arquitetura, executando o processo de síntese de cada componente separado e posicionando-os utilizando as ferramentas da Xilinx para tal propósito, como o PlanAhead. Evitando as tentativas automáticas das próprias ferramentas de mapeamento e roteamento de mapear e rotear toda a arquitetura como se fosse um único componente, tornando a tarefa muito mais complexa.

Para a arquitetura de grão grosso, sugere-se a implementação o algoritmo SPR, que foi verificado e validado em software, em uma versão em hardware, incluindo o escalonamento que pode ser integrado ao *Place & Route*, pois o mesmo já foi mostrado aqui que é viável ser

feito em hardware, quando foi utilizado na aplicação de bioinformática para roteamento da rede de interconexão da arquitetura.

8. Bibliografia

[Aldana, 2003] Aldana, M.; Boolean dynamics of networks with scale-free topology. Physica D 185:45–66. 2003.

[Ansari, 2009] Ansari A., Gupta S., Feng S., Mahlke S.; Zerehcache: armoring cache architectures in high defect density technologies. In MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 100–110, New York, NY, USA, 2009. ACM.

[Barabási, 1999] Barabási, A-L.; Albert, R.; Emergence of scaling in random networks. Science, 286:509-512, 1999.

[Barbie, 1999] Barbie, J., e Reblewski, F.; Emulation System having a Scalable Multi-level Multi-Stage Hybrid Programmable Interconnection Network, US Patent 5907679, 1999.

[Baumgarten ,2003] Baumgarten V.; PACT XPP – A Self-Reconfigurable Data Processing Architecture. The Journal of Supercomputing (TJS), 26(2): pp 167–184, 2003.

[Benes, 1965] Benes V. E.; Mathematical Theory of Connecting Networks and Telephone Traffic. Academic Press, New York, 1965.

[Bhattacharjya, 2006] Bhattacharjya, A.; Liang, S.; Median attractor and transients in random boolean nets. Physica D: Nonlinear Phenomena, vol 95, pp 29-34. 1996.

[Borkar,2006] Borkar. S.; Electronics beyond nano-scale cmos. In Design Automation Conference, 2006 43rd ACM/IEEE, pages 807 –808, 2006.

[Çam, 1999] Çam H.; Fortes J. A.; Work-efficient routing algorithms for rearrangeable symmetrical networks. IEEE Trans. Parallel Distrib. Syst., 10(7), 1999.

[Clos, 1953] Clos C.; A study of non-blocking switch networks. Technical report, Bell System Tech. J. 32:407-425, March, 1953.

[Coole, 2010] Coole J., Stitt G.; Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In IEEE/ACM CODES+ISSS, pages 13 – 22, 2010.

[Darabos , 2011] Darabos C., Tomassini M, Di Cunto F., Provero P., Moore J. H., Giacobini M.; Toward robust network based complex systems: from evolutionary cellular automata to biological models. Intelligenza Artificiale 5 DOI 10.3233/I-A-2011-0003, pages 37-47, 2011.

[DeHon, 2000] DeHon, A. ; Compact, multilayer layout for butterfly fat-tree In Proceedings of the 12th ACM symposium on Parallel algorithms and architectures, pp.206-215, 2000

[DeHone, 2002] DeHon A.; Very large scale spatial computing. In Unconventional Models of Computation, volume 2509 of Lecture Notes in Computer Science, pages 27–37. Springer Berlin / Heidelberg, 2002.

[Dinitz , 1999] Dinitz, Y., Even, S., Kupershtok, R., and Zapolotsky, M. ; Some compact layouts of the butterfly. In 11Th ACM Symposium on Parallel Algorithms and Architectures, 1999.

[Ejnioui, 1999] Ejnioui A., Ranganathan N.; Multi-terminal net routing for partial *crossbar*based multifpga systems. in ACM Int. Symposium on Field Programmable Gate Arrays, 1999.

[ExPRESS Benchmarks] ExPRESS Benchmarks, ECED, UCSB, USA, ”http://express.ece.ucsb.edu/benchmark/”, ultimo acesso em 20 de janeiro.

[FAPEMIG, 2011] FAPEMIG; Edital 2011; Redes Reguladoras de Genes com Topologia Livre de Escala utilizando Processamento Paralelo com Recursos Heterogêneos; coordenador: Ricardo dos Santos Ferreira. 2011.

[Ferreira, 2008] Ferreira R. S. ; Laure, M. ; Rutizig, M. ; Beck, A. C. ; Carro, L.; Reducing interconnection cost in coarse-grained dynamic computing through multistage network. IEEE International Conference on Field Programmable Logic and Applications (FPL), p. 47-52, 2008.

[Ferreira, 2009a] Ferreira, R. S. ; Laure, M. ; Lo T.; Rutizig, M. ; Beck, A. C. ; Carro, L.; A Low Cost and Adaptable Routing Network for Reconfigurable Systems. In: IEEE 16th Reconfigurable Architectures Workshop, 2009, rome, italy. IEEE Reconfigurable Architectures Workshop RAW09) - International Parallel & Distributed Processing Symposium (IPDPS), 2009.

[Ferreira, 2009b] Ferreira, R. S. ; Damiany A. ; Vendramini, J. ; Teixeira, T. ; Cardoso, J.; On Simplifying Placement and Routing by Extending Coarse-Grained Reconfigurable Arrays with Omega Networks. In: 5th International Workshop on Applied Reconfigurable Computing, 2009, Karlsruhe, Alemanha. International Workshop on Applied Reconfigurable Computing. Heidelberg, Alemanha : Springer-Verlag, 2009.

[Ferreira, 2011a] Ferreira, R. S., Vendramini, J., Mucida, L., Pereira, M., Carro, L.; An FPGA-based Heterogeneous Coarse-Grained Dynamically Reconfigurable Architecture. In: International Conference on Compilers, Architectures and Synthesis of Embedded Systems, 2011, Taiwan. ACM CASES - International Conference on Compilers, Architectures and Synthesis of Embedded Systems. USA : ACM, 2011.

[Ferreira, 20011b] Ferreira, R. S., Cardoso, João, Damiany (Alex), Vendramini, J. ; Teixeira, Tiago; Fast Placement and Routing by extending Coarse-Grained Reconfigurable Arrays with Omega Networks. Journal of Systems Architecture, v. 57, p. 761-777, 2011.

[Friedman, 2009] Friedman S., Carroll A., Van Essen B., Ylvisaker B., Ebeling C., Hauck S.; SPR: an architecture adaptive CGRA mapping tool. In Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays, FPGA ’09, pages 191–200, New York, NY, USA, AMC, 2009.

[Garg e Di Cara, 2008] Garg, A., Di Cara, A., Xenarios, I., Mendoza, L., De Micheli, G.; Synchronous versus asynchronous modeling of gene regulatory networks. Bioinformatics Systems Biology vol 24(17):1917-1925. 2008.

[Garg, 2008] Garg, A., Banerjee, D., De Micheli, G.; Implicit methods for probabilistic modeling of gene regulatory networks. International Conference of the IEEE Engineering in Medicine and Biology Society, pp. 1398-1404. 2008.

[Gazit, 1989] Gazit I.; On the number of permutations performable by extra-stage multistage interconnection networks. IEEE Trans. Comput., 38(2): pp 297–302, 1989.

[Geard, 1989] Geard, N.; Willadsen, K.; Dynamical approaches to modelling developmental gene regulatory networks. Birth Defects Research C: Embryo Today, 87 (2). pp. 131-142. 2009.

[Grammatikakis, 2001] Grammatikakis. M. D., Hsu D. F., Kraetzl M.; Parallel System Interconnections and Communications. CRC Press, 2001.

[Hauck, 2007] Hauck, S.; Dehon, A.; Reconfigurable Computing: The Theory And Practice Of Fpga-based Computation. Morgan Kaufmann, 2007.

[Hubner, 2006] Hubner, M.; Schuck, C.; Becker, J.; Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs. Parallel and Distributed Processing Symposium. IPDPS 2006. 20th International, 2006.

[Huelsbergen, 2000] Huelsbergen, L.; A representation for dynamic graphs in reconfigurable hardware and its application to fundamental graph algorithms. ACM/SIGDA Eighth international Symposium on Field Programmable Gate Arrays. 2000.

[Irons, 2006] Irons D. J.; Improving the efficiency of attractor cycle identification in boolean networks. Physica D, vol. 217, pp. 7–21, 2006.

[Kauffman,1969] Kauffman, S. A.; Metabolic stability and epigenesis in randomly constructed genetic nets. Journal of Theoretical Biology, 22:437-467. 1969.

[Kauffman, 1993] Kauffman, S. A.; Origins of Order: Self-Organization and Selection in Evolution. Oxford University Press. Technical monograph. ISBN 0-19-507951-5. 1993

[Knuth, 1963] Knuth, D.; The Art of Computer Programming, Vol. 2, Addison-Wesley, exercício 7 seção 3.1, pag. 7 e pag. 453. 1969.

[Lawrie, 1975] Lawrie, D.H.; Access and Alignment of Data in an Array Processor, IEEE Trans. on Computers, V24 (12), 1975.

[Lee, 1987] Lee K. Y.; A new benes network control algorithm. IEEE Trans. Comput., 36(6): pp.768–772, 1987.

[Leiserson, 1985] Leiserson, C. E.; Fat-trees: Universal networks for hardware efficient supercomputing. IEEE Transactions on Computers, v34 (10) p. 892–901, 1985.

[Lin, 1997] Lin,S., Lin,Y. e Hwang, T.; Net Assignment for the FPGA-Based Logic Emulation System in the Folded-Clos Network Structure, IEEE Trans. CAD, v16(3), 1997.

[Liu, 2008] Liu, Kevin K.; Comparing Throughput And Power Consumption In Both Sequential And Reconfigurable Processors. Trident Scholar Project rept. no. 369. 2008

[Lloyd, 2011] Lloyd S.; Snell Q.; Accelerated large-scale multiple sequence alignment. BMC Bioinformatics, 2011.

[Mei, 2003] Mei B., Vernalde S., Verkest D., Man H. D., Lauwereins R.; Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In DATE ’03: Proceedings of the conference on Design, Automation and Test in Europe, IEEE Computer Society, Washington, DC, USA, 2003.

[McLaughlin, 2009] McLaughlin K., O'Connor N., Sezer S.; Exploring cam design for network processing using fpga technology. In Advanced International Conference on Telecommunications AICT, pages 100–110, New York, NY, USA, ACM, 2009.

[Micheli, 1994] Micheli G. D.; Synthesis and Optimization of Digital Circuits, 1st ed. McGraw-Hill Higher Education, 1994.

[ML605 web site] ML605 web site; http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm último acesso, 20 de fevereiro de 2012.

[Mohan, 2006] Mohan N., Fung W., Sachdev M.; Low power priority encoder and multiple match detection circuit for ternary content addressable memory. In Advanced International Conference on Telecommunications AICT, 2006.

[Nassami, 1981] Nassimi D., Sahni S.; A self-routing benes network and parallel permutation algorithms. IEEE Trans. Comput., 30(5): pp.332–340, 1981.

[Neji, 2008] Neji B., Aydi Y., Ben-atilallah R., Meftaly S., Abid M.; Multistage interconnection network for mpsoc: Performances study and prototyping on fpga. In IEEE Design and Test Workshop (IDT), 2008.

[Park, 2008] Park. Song-Jun., Shires, D., Henz. B.; Reconfigurable Computing: Experiences and Methodologies. A263874, 2008.

[Pournara, 2005] Pournara I., Bouganis C.S., Constantinides G.A.; Fpga-accelerated bayesian learning for reconstruction of gene regulatory networks. IEEE Field Programmable Logic Conference (FPL) - Pages 323—328. 2005.

[Schäck, 2009] Schäck C., Heenes W., Hoffmann R.; A multiprocessor architecture with an omega network for the massively parallel model gca. In Embedded Computer Systems: Architectures, Modeling, and Simulation, volume 5657 of Lecture Notes in Computer Science, pages 98–107. Springer Berlin / Heidelberg, 2009.

[Tagkopoulos, 2003] Tagkopoulos, I., Zukowski, C., Cavelier, G., and Anastassiou, D.; A custom FPGA for the simulation of gene regulatory networks. In Proceedings of the 13th ACM Great Lakes Symposium on VLSI. 2003.

[Tanigawa, 2008} Tanigawa K., Zuyama T., Uchida T., Hironaka T.; Exploring compact design on high throughput coarse grained reconfigurable architectures, in IEEE International Conference on Field Programmable Logic and Applications (FPL), pp. 126–135,2008.

[Vendramini, 2010a] Vendramini J. C. G., Ferreira R.; Redes de interconexão multiestágios em plataformas reconfiguráveis para sistemas embarcados. In Workshop de Sistemas Embarcados, Gramado, Brasil, 2010.

[Vendramini, 2010b] Vendramini, J., Ferreira, R. S.; Parallel Routing Algorithm for Extra Level Omega Networks on Reconfigurable Systems. In: XI Simpósio de Sistemas Computacionais (WSCAD-SSC), Petrópolis, 2010.

[Watts, 1998] Watts, D.J. ; Strogatz, S.H.; Collective dynamics of small-world networks.! Nature, 393:440-442. 1998

[Wu, 1980] C-L., Feng, T-Y.; On a Class of Multistage Interconnection Networks. IEEE Trans. Comput. V29 (8), 694-702, 1980.

[Xia, 2011] Xia, F.; Dou Y.; Lei G.; Tan Y.; FPGA accelerator for protein secondary structure prediction based on the GOR algorithm. BMC Bioinformatics, 2011.

[Xilinx Web Site] Xilinx Web Site; "http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm", último acesso em 22 de fevereiro de 2012.

[Yeh, 1992] Yeh, Y-M., e Tse-yun Feng, T-Y; On a Class of Rearrangeable Networks, IEEE Trans. on Computers, v41(11), pp. 1361—1379, 1992.

[Yoon, 2009] Yoon J. W., Shrivastava A., Park S., Ahn M., Paek Y.; A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. IEEE Trans. Very Large Scale Integr. Syst., 17:1565–1578, November 2009.

[Zerarka, 2004] Zerarka, M., David, J.P., Aboulhamid, E. M.; High speed Emulation of gene Regulatory Networks using FPGAs. 47th IEEE International MidWest Symposium on Circuits and Systems, pp-545-548. 2004.

[Zhang, 2007] Zhang S. Q., Hayashida M., Akutsu T., Ching W. K.; Algorithms for finding small attractors in Boolean networks. Eurasip Journal on Bioinformatics and System Biology, pp. 1–13, 2007.